INTRODUCTION

The MC68881 floating-point coprocessor is a complete implementation of the proposed IEEE Standard for Binary Floating-Point Arithmetic (Task P754, Draft 10.0). All data formats, data types, operations, modes, conversions, and exception handling required in a conforming implementation of the proposed standard are supported entirely in hardware. Additionally, a full library of fast elementary transcendental functions is implemented in the hardware.

The MC68881 is a high performance floating-point unit designed to interface with the 32-bit MC68020 as a coprocessor; it provides a logical extension of the MC68020 instruction set and register set in a manner which is completely transparent to the programmer. The MC68020 microprocessor implements the M68000 coprocessor interface entirely in hardware. All interprocessor transfers are initiated by the MC68020. During the processing of an MC68881 instruction, the MC68020 transfers instruction information and data to the coprocessor via standard M68000 write bus cycles and receives data, requests for service, and status information from the coprocessor via standard M68000 read bus cycles. The MC68881 contains a number of coprocessor interface registers which are addressed like memory by the MC68020 microprocessor. These registers, which are not part of the MC68881 programmer-visible register set, are mapped into CPU address space in an MC68020 system. The MC68020 distinguishes CPU address space accesses from program and data accesses by a unique function code encoding. (The MC68881 registers that are used by the floating-point instructions and hence visible to the programmer are the floating-point registers, FPO-FP7, and the control, status, and instruction address registers.)

The MC68881 can also be used as a peripheral processor in systems where the main processor does not implement the M68000 coprocessor interface on-chip (e.g., MC68000, MC68008, and MC68010 systems). Since the coprocessor interface is based solely on standard M68000 bus cycles, it is easily emulated by software in these systems. The MC68881 is considered to be a peripheral processor in these systems because its coprocessor interface registers are memory-mapped in data address space. Two methods of software emulation of the coprocessor interface are possible: 1) M68000 F-line instruction trap (traps are exceptions caused by instructions), or 2) in-line code implemented as either subroutine calls or macros.

When assembled for execution in an MC68020 system, the first word of an MC68881 instruction always has a hexadecimal F (binary 1111) in the most significant nibble (Figure 1). When MC68000, MC68008, or MC68010 processors encounter an F-line instruction, the current processor status is saved, the F-line emulation trap vector is fetched, and instruction execution resumes in the trap handler. When this trap handler is a software emulation of the coprocessor interface, object code containing MC68881 instructions is upward compatible to an MC68020 system without recompiling, reassembling, or relinking. MC68881 instruction performance will significantly increase when such code is ported to a MC68020 system where the coprocessor interface is implemented by on-chip hardware. However, a performance penalty is paid for the upward compatibility provided by the F-line emulation trap method.

The current processor status (either three or four words depending upon the M68000 processor being used) must be pushed onto the stack when the exception is taken, and popped off of the stack when the RTE (return from exception) instruction is executed. More significantly, the trap handler must decode the MC68881 instruction to determine the proper protocol for a given instruction. The performance penalty can become intolerable if the coprocessor emulation trap handler must support all M68000 effective addressing modes. This can be relieved by utilizing only specific addressing modes.

Upward compatibility of object code is not relevant to every M68000 system, especially when it incurs a significant performance penalty. Such systems can emulate the coprocessor interface using in-line code, macros, or subroutine calls. Macros are particularly attractive since they provide the performance of in-line code while allowing the source code to be upward compatible to an MC68020 system via re-compilation or reassembly. This application note provides both a discussion of the coprocessor interface protocol followed by detailed information on both a macro technique and an example of an F-line instruction trap technique for software emulation of the coprocessor interface.

MC68881 as an MC68020 System Coprocessor

COPROCESSOR CONCEPT

The M68000 coprocessor interface is the first general purpose coprocessor interface. The main processor instruction set and internal details are unknown to the coprocessor, and the coprocessor instruction set and internal details are unknown to the main processor. The burden
of decoding the coprocessor instruction is not placed on the main processor, and the coprocessor does not monitor the main processor instruction stream.

The MC68881 instruction set and register set are logical extensions of the MC68020 sets. The MC68020/MC68881 execution model appears to the programmer as if implemented on one chip. All data transfers are performed by the main processor at the request of the MC68881; thus, memory management, bus errors, address errors, and bus arbitration all function as if MC68881 instructions were executed by the main processor. The main processor also performs all effective address calculations and processes all coprocessor-detected exceptions at the request of the MC68881.

The interface is designed to provide the programmer with a sequential instruction execution model even though main processor and coprocessor instruction execution overlap occurs. For some instructions this overlap to enhances throughput.

MC68020 INTERPROCESSOR BUS TRANSFERS

The coprocessor interface is based upon standard M68000 asynchronous read and write bus cycles. No new bus signals are required by the MC68020 to initiate a floating-point instruction, and the MC68020 and MC68881 may run at different clock rates. The MC68881 is connected like a peripheral to the main processor and requires one extra pin, chip select (CS), in order to be accessed. Chip select is generated from the MC68020 at different clock rates. The MC68881 is configured to operate over a 16-bit data bus. The MC68881 chip select is therefore based upon three elements: the MC68020 three function code outputs, the Cp-ID field (address bits 15-13 of the address bus), and the CPU space type field (bits 19-16 of the address bus).

Notice, the MC68020 handles only four CPU address space cycles:

<table>
<thead>
<tr>
<th>CPU Space Type Field (A19-A16)</th>
<th>CPU Space Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Breakpoint Acknowledge</td>
</tr>
<tr>
<td>0001</td>
<td>Access Level Control</td>
</tr>
<tr>
<td>0010</td>
<td>Coprocessor Communications</td>
</tr>
<tr>
<td>1111</td>
<td>Interrupt Acknowledge</td>
</tr>
</tbody>
</table>

Therefore, when decoding the chip select for the MC68881, only two bits are needed (A18 and A17) to distinguish a coprocessor operation from the other CPU address space operations. A suggested method for connecting the MC68881 to the MC68020 is illustrated in Figure 4.

Figure 5 illustrates the connection of an MC68881 to an MC68000 or MC68010 as a peripheral processor over a 16-bit data bus. The MC68881 is configured to operate over a 16-bit data bus when the size pin is connected to VCC, and the A0 pin is connected to ground (GND). The sixteen least significant data pins (D0-D15) must be connected to the sixteen most-significant data pins (D16-D31) when the MC68881 is configured to operate over a 16-bit bus (i.e., connect D0 to D16, D1 to D17,...,and D15 to D31). The DSACK1 pin of the MC68881 is connected to the DTACK pin of the main processor, and the DSACK0 pin is not used.
Figure 2. Coprocessor CPU Space Decoding (Function Code = 111)

<table>
<thead>
<tr>
<th>Address Bits 6-15</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$000</td>
<td>RESPONSE (R)*</td>
</tr>
<tr>
<td>$002</td>
<td>SAVE (R)*</td>
</tr>
<tr>
<td>$004</td>
<td>OPWORD (R/W)</td>
</tr>
<tr>
<td>$006</td>
<td>(RESERVED)</td>
</tr>
<tr>
<td>$008</td>
<td>RESTORE (R/W)*</td>
</tr>
<tr>
<td>$00C</td>
<td>COMMAND (W)*</td>
</tr>
<tr>
<td>$010</td>
<td>CONDITION (W)*</td>
</tr>
<tr>
<td>$012</td>
<td>OPERAND (R/W)*</td>
</tr>
<tr>
<td>$014</td>
<td>REGISTER SELECTOR (R)**</td>
</tr>
<tr>
<td>$016</td>
<td>(RESERVED)</td>
</tr>
<tr>
<td>$018</td>
<td>INSTRUCTION ADDRESS (W)*</td>
</tr>
</tbody>
</table>

*Denotes required registers for minimum support of all coprocessor instruction types. The MC68881 implements these registers.
**Denotes additional registers implemented by the MC68881.

Figure 3. Coprocessor Interface Registers

Figure 4. Suggested MC68020 to MC68881 Connections
When connected as a peripheral processor, the MC68881 chip select (CS) decode is system dependent. If the MC68000 is used as the main processor, the MC68881 CS must be decoded in the supervisor or user data spaces. However, if the MC68010 is used for the main processor, the MOVES instruction may be used to emulate any CPU space access that the MC68020 generates for coprocessor communications. Thus, the CS decode logic for such systems may be the same as in an MC68020 system, such that the MC68881 will not use any part of the address spaces.

Figure 5. 16-Bit Data Bus Peripheral Processor Connection

Figure 6 illustrates the connection of an MC68881 to an MC68008 as a peripheral processor over an 8-bit data bus. The MC68881 is configured to operate over an 8-bit data bus when the SIZE pin is connected to ground (GND). The eight least-significant data pins (D0-D7) must be connected to the twenty-four most-significant data pins (D8-D31) when the MC68881 is configured to operate over an 8-bit data bus (i.e., connect D0 to D8, D16, and D24; D1 to D9, D17, and D25; ... D7 to D15, D23, and D31). The DSACK0 pin of the MC68881 is connected to the DTACK pin of the MC68008, and the DSACK1 pin is not used.

Figure 6. 8-Bit Data Bus Peripheral Processor Condition
When connected as a peripheral processor, the MC68881 chip select (CS) decode is system dependent, and the CS must be decoded in the supervisor or user data spaces.

**MC68881 INSTRUCTION DEFINITION**

The MC68881 instructions can be subdivided according to the type of coprocessor operation performed: general, branch, save, restore, or conditional. Each instruction, when assembled, consists of from one to eight words (Figure 7). The first word (operation word) always has a hexadecimal F (1111) in the high-order nibble as seen in Figure 1. The type field (bits 8-6) of the operation word indicates the coprocessor instruction type. For instruction types which require an effective address (general, save, restore, and conditional), the type-dependent field of the operation word specifies the effective addressing mode. For the conditional instruction type, this field specifies the condition to be evaluated by the coprocessor: the conditional predicate (CPRED).

![Table 1. General Format of Coprocessor Instruction](image)

The format of the second word of the coprocessor instruction varies with different instruction types. For the general instructions, the second word is the coprocessor command word specifying the function to be performed by the coprocessor. The MC68881 has been designed such that all general type operations are specified by a single command word. In order to process a conditional instruction type, the main processor must deliver the conditional predicate (CPRED) to the MC68881 for evaluation. Since the type-dependent field of the operation word may specify the effective addressing mode, the CPRED is found in the low-order six bits of the second word. The additional word(s) following the second word in Figure 8 is the extension to the effective addressing mode or the immediate operands present in the instruction. In the branch, save, and restore instructions, all information needed to initiate processing in the coprocessor is found in the operation word. Thus, the extension word(s) (if any) directly follows the operation word (no coprocessor command word).

![Figure 7. General Format of Coprocessor Instruction](image)

The MC68881 instruction and ask the main processor to query again.

The restore type instruction which restores a previously saved state is initiated by the main processor writing the coprocessor command word (the second word of the instruction) to the MC68881 coprocessor interface command register. The format of the MC68881 command word is shown in Figure 8. General type instructions are broken down into groups, called op-classes, based on the function of the instruction and argument location (external or internal to the coprocessor). The values Rx, Ry, and the extension field depend on the specific op-class. For instance, the values required for a floating-point register to floating-point register operation are as follows: Rx is the source floating-point register, Ry is the destination floating-point register, and the extension field is the operation to be performed (add, move, sin, etc.). Table 1 lists the op-classes, their definitions, and their respective Rx, Ry, and extension fields.

For the branch and the conditional type instructions, the main processor initiates processing by writing the conditional predicate (CPRED) from the six low-order bits of either the first or second word of the instruction, respectively, to the coprocessor for evaluation. These conditional predicates are found on lines 62-97 of the EQUATE table listed in **APPENDIX A MACROS** at the back of this document.

In the case of an operating system context switch, the coprocessor internal state can be saved by the FSAVE instruction. This MC68881 instruction only saves the invisible state of the machine (that which is not normally available to the user). Thus all control, status, instruction address, and floating-point data registers (the user-visible registers) must be saved by the user. Only the registers beneficial (those being used) to the programmer need to be stored. To initiate the FSAVE instruction sequence, the main processor reads a word (the save format word) from the save interface register location of the coprocessor. The save format word provides the status of the coprocessor state (the null state, the idle state, or the busy state) and also the size (0 bytes, 24 bytes, or 180 bytes respectively) of the internal state of the machine to be saved. The save format word is written to the effective address by the main processor at the end of the instruction execution no matter which state the coprocessor is in.

The restore type instruction which restores a previously saved state is initiated by the main processor writing a save format word to the coprocessor interface restore register. This informs the coprocessor which coprocessor internal state is to be restored. If visible registers are saved after the execution of the FSAVE command, they must be restored before the execution of the FRESTORE instruction.

When executing any MC68881 instruction, the MC68020 follows a basic protocol. First, the coprocessor information (command word, conditional predicate, or format word) is written to the appropriate coprocessor interface register by the main processor (the FSAVE instruction is initiated by a read). The main processor then reads the appropriate coprocessor interface register to acquire the coprocessor status and any service requests. The coprocessor may indicate that it is busy processing a previous instruction and ask the main processor to query again. (This is the mechanism by which the sequential instruction execution is maintained because the coprocessor...
### Table 1. Command Word Fields of General-Type Instructions

<table>
<thead>
<tr>
<th>Op-Class Field</th>
<th>Rx Field</th>
<th>Ry Field</th>
<th>Extension Field</th>
<th>Instruction Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Source FP Data Register</td>
<td>Destination FP Data Register</td>
<td>Operation to Perform (MOVE, ADD, etc.)</td>
<td>FP Data Register to FP Data Register Instructions</td>
</tr>
<tr>
<td>001</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Unused, Reserved</td>
</tr>
<tr>
<td>010</td>
<td>Source Data Format (see Note 2)</td>
<td>Destination FP Data Register</td>
<td>Operation to Perform (MOVE, ADD, etc.)</td>
<td>External Operand to FP Data Register</td>
</tr>
<tr>
<td>010</td>
<td>111</td>
<td>Destination FP Data Register</td>
<td>Constant ROM Offset</td>
<td>Move Constant to FP Data Register</td>
</tr>
<tr>
<td>011</td>
<td>Destination Data Format (see Note 2)</td>
<td>Source FP Data Register</td>
<td>00000000 (Unless Packed BCD, see Note 2)</td>
<td>Move out FP to External Destination</td>
</tr>
<tr>
<td>100</td>
<td>FPCr Select (see Note 1)</td>
<td>000</td>
<td>00000000</td>
<td>Move/Move Multiple to Control Register</td>
</tr>
<tr>
<td>101</td>
<td>FPCr Select (see Note 1)</td>
<td>000</td>
<td>00000000</td>
<td>Move/Move Multiple from Control Register</td>
</tr>
<tr>
<td>110</td>
<td>A/D S/D 0 (see Note 3)</td>
<td>00m (see Note 4)</td>
<td>Contains Register List</td>
<td>Move Multiple FP Data Register to MC68881</td>
</tr>
<tr>
<td>111</td>
<td>A/D S/D 0 (see Note 3)</td>
<td>00m (see Note 4)</td>
<td>Contains Register List</td>
<td>Move Multiple FP Data Register from MC68881</td>
</tr>
</tbody>
</table>

**NOTES:**

1. **FPCr** Floating-Point Control Register
   - 000: Reserved
   - 001: FPIAR (Instruction Address Register)
   - 010: FPSR (Status Register)
   - 011: FPSR and FPIAR
   - 100: FPCR (Control Register)
   - 101: FPCR and FPIAR
   - 110: FPCR and FPSR
   - 111: FPCR, FPSR, and FPAIR

2. **Value**
   - 000: Long Word Integer
   - 001: Single Precision Real
   - 010: Extended Precision Real
   - 011: Packed Decimal Real
   - 100: Word Integer
   - 101: Double Precision Real
   - 110: Byte Integer
   - 111: For Op-Class = 011, Packed Decimal Real with Dynamic k Factor specified as CPU Data Register by Extension Field Encoding, rrr0000

3. **A/D = 0** Most Significant Bit of Register List Selects FP7
   - **A/D = 1** Most Significant Bit of Register List Selects FP0
   - **S/D = 0** Bit Mask in Extension Field
   - **S/D = 1** Bit Mask in CPU Data Register Selected by Extension Field, 0rr0000

4. When Bit Mask is Transferred to Command Word: "m" is the Most-Significant Bit of the Register List

---

**INTERFACE REGISTERS**

The MC68881 contains a number of interface registers which are memory-mapped within the MC68020 CPU address space. These are the registers identified by "**" and "***" in Figure 8. The coprocessor registers are memory-mapped into the main processor's system like any other peripheral, although they are accessed in a different address space.

The main processor initiates communication with the MC68881 by writing to (or reading from) a specific 16-bit
register, which is memory-mapped into the system. The specific register chosen depends on the instruction type. A general instruction coprocessor command word is written to the command register, and a branch or conditional instruction CPRED is written to the condition register. The main processor must read the save interface register to initiate the sequence of saving the internal state. To restore this state, the main processor writes to the restore interface register. The save and restore functions support virtual memory, demand paging, and task time-slicing.

After the initial write to the register in the general, conditional, and branch instructions, the response register is read by the processor to determine its next action (e.g., come-again or evaluate effective address and transfer data to/from the coprocessor).

An example of the communication sequence may be demonstrated with the memory to floating-point register add instruction. The main processor first writes the coprocessor command to the command register and seeks the response register until requested to pass data. At which time, the host reads the data from memory and writes it to the operand register, four bytes at a time. The response register is re-read until the coprocessor signals the main processor to stop. The MC68020 is then free to process the next main processor instruction, while the MC68881 performs the floating-point add on the data. In this case, as in all normal processor/coprocessor communications, the host processor processes only the requests specified in the coprocessor primitives until released by the MC68881.

The MC68881 uses only three registers other than those previously mentioned. These are: the register select register used in the move multiple instructions, the instruction address register used only when exceptions are enabled, and the control register used by the main processor either to handshake the processing of the coprocessor exception or to abort invalid coprocessor service requests.

**RESPONSE PRIMITIVES**

Response primitives are service requests from the coprocessor to the main processor. These primitives have capabilities which allow for the synchronization of the main processor/coprocessor general, conditional, and branch instruction executions. Also the coprocessor may request services such as external memory accesses, transfer of data, and exception processing by the main processor. Figure 9 is a list of all possible coprocessor response primitives recognized by the MC68020. If the come-again (CA) bit is set, the main processor processes the primitive and then reads the response register again to seek further service requests. If the CA bit is not set, the main processor is released from further services (except when the MC68020 is in trace mode). If the PC bit is set, the main processor writes the program counter position of the first word of the coprocessor instruction to the instruction address interface register prior to performing the requested service. In the event that the MC68881 generated a trap exception, this PC value is required by the exception trap handler to determine which instruction caused the exception.

The MC68881 utilizes only six of the possible responses. These are the primitives noted by an "*" in Figure 9. The transfer multiple coprocessor registers primitive allows the transfer of multiple coprocessor registers to or from memory. The dr bit specifies in which direction the transfer is to be made: from the coprocessor to memory if the bit is set, or from the memory into the coprocessor if the bit is cleared. The transfer single main processor register requests the main processor to transfer the contents of one of its registers to or from the coprocessor. If D/A equals one, the transferred register is an address register. If D/A equals zero, a data register is transferred. The register number is located in the register field.

The evaluate effective address and transfer data primitive requests the main processor to evaluate the effective address specified by the instruction and to transfer data to or from that address (from or to the coprocessor). The valid EA type field indicates which addressing modes are valid for the transfer, while the length field gives the number of bytes to be transferred.

The Null primitive alerts the main processor to the coprocessor status after all other service requests (excluding exception requests) have been granted by the main processor. If the CA bit is set, the main processor queries the response register until the bit is cleared, at which time the main processor is released by the coprocessor. Even though the main processor may be signaled for release (when CA equals zero), it can still pass the program counter (PC equals one) and accept pending interrupts (IA equals one and CA equals one). The processing finished (PF) bit is a status bit which indicates whether or not the coprocessor has finished its instruction. The MC68020 tests the bit only while in trace mode to ensure that the instruction processing is complete. In the case of a conditional instruction the null primitive also contains a T/F bit (bit 0). This bit is tested by the main processor to determine whether or not the conditional predicate is true (one) or false (zero).

For all the MC68881 primitive responses, the CA bit is always set (CA equals one) with the exception of the null and exception request primitives. Both the take pre-instruction exception and the take mid-instruction exception primitives contain the exception vector number. Pre-instruction exceptions occur under two conditions: 1) after no further information is needed from the main processor in a previous floating-point instruction, and 2) before the coprocessor begins processing the present instruction. These pre-existing exceptions represent either an illegal command word for the present instruction, or the termination of a previous instruction with an exception. This delayed reporting allows for synchronization between the host and the coprocessor in the event of any pre-existing exceptions. A floating-point register to memory move (op-class 001) operation is the only instruction capable of generating a mid-exception primitive. It is detected in the last read of the response register during instruction execution because the MC68881 performs the floating-point calculation and releases the main processor only after the data transfer to memory. With the memory to floating-point register (op-class 010) or the floating-point register to floating-point register (op-class 000), the main
<table>
<thead>
<tr>
<th>BUSY</th>
<th>TRANSFER MULTIPLE COPROCESSOR REGISTERS</th>
<th>TRANSFER STATUS AND SCANPC</th>
<th>SUPERVISOR CHECK</th>
<th>TAKE ADDRESS AND TRANSFER DATA</th>
<th>*TRANSFER MULTIPLE MAIN PROCESSOR REGISTERS</th>
<th>TRANSFER OPERATION WORD</th>
<th>*NULL</th>
<th>EVALUATE EFFECTIVE ADDRESS AND TRANSFER ADDRESS</th>
<th>*TRANSFER SINGLE MAIN PROCESSOR REGISTER</th>
<th>TRANSFER MAIN PROCESSOR CONTROL REGISTER</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt=" BUSY Code" /></td>
<td><img src="image" alt=" TRANSFER MULTIPLE COPROCESSOR REGISTERS Code" /></td>
<td><img src="image" alt=" TRANSFER STATUS AND SCANPC Code" /></td>
<td><img src="image" alt=" SUPERVISOR CHECK Code" /></td>
<td><img src="image" alt=" TAKE ADDRESS AND TRANSFER DATA Code" /></td>
<td><img src="image" alt=" *TRANSFER MULTIPLE MAIN PROCESSOR REGISTERS Code" /></td>
<td><img src="image" alt=" TRANSFER OPERATION WORD Code" /></td>
<td><img src="image" alt=" *NULL Code" /></td>
<td><img src="image" alt=" EVALUATE EFFECTIVE ADDRESS AND TRANSFER ADDRESS Code" /></td>
<td><img src="image" alt=" *TRANSFER SINGLE MAIN PROCESSOR REGISTER Code" /></td>
<td><img src="image" alt=" TRANSFER MAIN PROCESSOR CONTROL REGISTER Code" /></td>
</tr>
</tbody>
</table>

Figure 9. Coprocessor Response Primitives (Sheet 1 of 2)
TRANSFER TOP-OF-STACK

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>PC</td>
<td>dr</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

TRANSFER FROM INSTRUCTION STREAM

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>PC</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

*EVALUATE EFFECTIVE ADDRESS AND TRANSFER DATA

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>PC</td>
<td>dr</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

*TAKE PRE-INSTRUCTION EXCEPTION

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PC</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

*TAKE MID-INSTRUCTION EXCEPTION

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PC</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

TAKE POST-INSTRUCTION EXCEPTION

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PC</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

INVALID (RESERVED)

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>PC</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

WRITE TO PREVIOUSLY EVALUATED EFFECTIVE ADDRESS

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>PC</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

INVALID (RESERVED)

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>PC</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTES:
- `dr`: 0 = Into Coprocessor
  1 = Out of Coprocessor
- `D/A`: 0 = Data
  1 = Address
- `Valid EA Types:`
  000 = Control Alterable
  001 = Data Alterable
  010 = Memory Alterable
  011 = Alterable
  100 = Control
  101 = Data
  110 = Memory
  111 = Any Mode Allowed (No Restrictions)

*Supported by MC68881

Figure 9. Coprocessor Response Primitives (Sheet 2 of 2)
processor is freed to execute the next instruction as the coprocessor performs the requested operation. The save, restore, and move multiple instructions do not generate exceptions.

SOFTWARE TO EMULATE THE COPROCESSOR INTERFACE

DEFINITIONS AND ASSUMPTIONS

In order to utilize the floating-point coprocessor in a MC68000/MC68008/MC68010 system, a software emulation of the coprocessor interface must be developed. There are two possible methods of software emulation: 1) in-line code such as macros or subroutine calls, or 2) the M68000 F-line emulation trap.

The coprocessor must reside in a different address space than in a MC68020 system. When the MC68020 accesses the coprocessor, it does so in CPU space by outputting a 111 on the function code lines. The equivalent function codes generated on either MC68000, MC68008, or MC68010 signify an interrupt acknowledge bus cycle, i.e. these processors do not implement CPU space. Thus the MC68881 must be accessed as a peripheral with the coprocessor interface registers memory-mapped in data address space in these systems, not in CPU space.

To accommodate the practical use of this application, the demonstration software will perform a floating-point instruction in the fastest way possible while not violating the safety provided in the IEEE standard and the MC6881.

To do this, a number of factors are first considered. The interprocessor protocol used with all instruction type classes may be minimized to include only those checking mechanisms necessary to perform the basic function. The most significant simplification is to restrict the use of the ENABLE byte of the floating-point control register. Each bit of this byte represents a type of coprocessor detected exception. Figure 10 identifies all coprocessor detected exceptions, their corresponding vector number (passed to the MC68881 in an exception primitive), and their position in the ENABLE byte. Two exceptions are not represented in this byte: the protocol violation and the illegal coprocessor command. This demonstration software ignores all take exception responses from the MC68881 (generated by the conditions specified in the ENABLE byte) which reduces the overhead required for their recognition in the software. An exception primitive, due to a floating-point operation, will be generated only if the MC68881 records an exception and the corresponding bit in the ENABLE byte is also set. Thus an exception primitive cannot be generated if the default condition (SO0) in the ENABLE byte is never altered by the user. This eliminates the need for monitoring the response register for pre-instruction or mid-instruction exceptions. If a coprocessor-detected exception occurs, it will never be detected by the main processor.

Note that two coprocessor detected exceptions, the protocol violation and the illegal coprocessor command, are not represented by a bit in the ENABLE byte. A protocol violation occurs anytime communication between the main processor and coprocessor is improper and is reported by the MC68881 as a mid-instruction exception. An illegal coprocessor command is a coprocessor command not implemented by the MC68881 and is reported as a pre-instruction exception. Therefore, by not checking for exceptions, a protocol violation or illegal coprocessor command may occur without being detected. Another consequence of the previously mentioned simplification is that the program counter will never be requested of the main processor if exceptions are disabled. The PC bit of the responses will never be set if the bits in the ENABLE byte are not set. Therefore, the overhead for testing of this bit can be saved.

<table>
<thead>
<tr>
<th>ENABLE EXCEPTION BIT</th>
<th>VECTOR DEC</th>
<th>NUMBER HEX</th>
<th>VECTOR DEC</th>
<th>OFFSET HEX</th>
<th>ASSIGNMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSUN</td>
<td>48</td>
<td>$30</td>
<td>102</td>
<td>$0C0</td>
<td>BRANCH OR SET ON UNORDERED CONDITION</td>
</tr>
<tr>
<td>INEX2/INEX1</td>
<td>49</td>
<td>$31</td>
<td>196</td>
<td>$0C4</td>
<td>INEXACT RESULT</td>
</tr>
<tr>
<td>DZ</td>
<td>50</td>
<td>$32</td>
<td>200</td>
<td>$0C8</td>
<td>FLOATING-POINT DIVIDE BY ZERO</td>
</tr>
<tr>
<td>UNFL</td>
<td>51</td>
<td>$33</td>
<td>204</td>
<td>$0CC</td>
<td>UNDERFLOW</td>
</tr>
<tr>
<td>OPERR</td>
<td>52</td>
<td>$34</td>
<td>208</td>
<td>$0D0</td>
<td>OPERAND ERROR</td>
</tr>
<tr>
<td>OVFL</td>
<td>53</td>
<td>$35</td>
<td>212</td>
<td>$0D4</td>
<td>OVERFLOW</td>
</tr>
<tr>
<td>SNAN</td>
<td>54</td>
<td>$36</td>
<td>216</td>
<td>$0D8</td>
<td>SIGNALING NAN</td>
</tr>
<tr>
<td>NONE</td>
<td>11</td>
<td>$0B</td>
<td>44</td>
<td>$02C</td>
<td>F-LINE EMULATOR</td>
</tr>
<tr>
<td>NONE</td>
<td>13</td>
<td>$0D</td>
<td>52</td>
<td>$034</td>
<td>COPROCESSOR PROTOCOL VIOLATION</td>
</tr>
</tbody>
</table>

Figure 10. MC68881 ENABLE Byte and Coprocessor Exceptions
IN-LINE CODE, MACROS, OR SUBROUTINES

The first approach discussed is the one to directly drive the MC68881 as a peripheral from the user program in user data space. This approach is used when speed of the MC68881 instruction execution is more important than upward compatibility of the object code. Two methods are available to drive the peripheral in user data space: in-line code (macros) or run-time libraries (subroutines). The trade-off between the two approaches concerns time versus space. Each time a macro is used the length of the source code increases by the size of the macro. When a subroutine is called, the overhead of the subroutine call and the execution of the RTS instruction must be incurred. No F-line trap is taken in either method, which saves the time to perform the corresponding stacking and instruction decoding.

Macros allow the coding of a repeated pattern of instructions which may contain variable entries at each iteration of the pattern. By incorporating the use of macros with conditional assembly instructions, some of the necessary floating-point instruction decode can be done during assembly time, reducing the run-time overhead. Assuming that the assembler used has the ability to pass parameters and manipulate them within the macro during assembly, the programmer need not generate the code to parse the floating-point instruction to detect the addressing mode used — it is passed directly by the macro into the assembled code. Subroutines can also act as an extension to the in-line routine to perform functions common to each instruction type (exception handling routines, error checking routines, etc.). The major advantages of using these methods over the F-line trap approach are: 1) the time saved by not taking the F-line trap, and 2) the time saved by assembly time instruction decode.

This application note includes software for the macro approach to drive the code in user data space. The same code would be pertinent if implementing a run-time library approach.

There are different ways to define the macros to drive the MC68881 depending on the particular application. The ideal method would be to write a macro supporting each floating-point instruction employed to achieve source code compatibility with the MC68020. In order to support the complete MC68881 instruction set, the library of macro definitions would be quite large. For conciseness, multiple coprocessor instructions are consolidated into single macro instructions, collected by the method of operand transfer required of the instruction (located in APPENDIX A MACROS). As an example, all MOVE-INS, MOVE-OUTS, etc. have their own macros corresponding to their respective directions and precisions. Most MC68881 instructions are supported, but source code compatibility with the standard MC68881 instruction set is lost due to consolidation of specific instructions into single macros. The conditional trap and the move multiple coprocessor system register instructions are not included in this example set.

Every macro requires at least the same amount of information supplied by the programmer as the represented MC68881 floating-point instruction and, in some cases, more information. This data is passed to the macro by parameters. An example of a general instruction class macro call is:

MACRONAME FUNCTION, SOURCE, DESTINATION

MACRONAME specifies the method of operand transfer, the FUNCTION is the general operation to be performed, the SOURCE is the location of the source operand, and the DESTINATION is the location of the destination operand. Different macros request different information of the user. For instance, the FSAVE instruction only requires one parameter to be passed to the macro. Basically, each macro follows a similar format and is described in Functional Description. The methodology of these macro definitions can be employed by the programmer who wishes to develop a separate macro for each instruction.

Functional Description

The following paragraphs provide the programmer with information on how to use the macros. All of the macros for the general instruction which transfer a source operand to the coprocessor (move-in) are listed below:

MEMREGB Function, SourceEA, FPn
MEMREGW Function, SourceEA, FPn
MEMREGL Function, SourceEA, FPn
MEMREGS Function, SourceEA, FPn
MEMREGD Function, (An), FPn
MEMREGX Function, (An), FPn
MEMREGP Function, (An), FPn

They only differ by the precision of the data transferred. Each macro of this class transfers a source operand (specified by SourceEA or (An)) of a specific precision to the MC68881 and performs the operation specified by function. A list of the functions describing the supported MC68881 instructions and their functions can be found on lines 25-61 of the EQUATE table of the demonstration software found in APPENDIX A MACROS.

The macros which transfer data from a single floating-point register to an effective address (move out) are:

REGMEMB FMOVE, FPm, DestinationEA
REGMEMW FMOVE, FPm, DestinationEA
REGMEML FMOVE, FPm, DestinationEA
REGMEMS FMOVE, FPm, DestinationEA
REGMEMD FMOVE, FPm, (An)
REGMEMX FMOVE, FPm, (An)
REGMEMP FMOVE, FPm, (An), [k-factor]

The MC68881 FMOVE command is the only MC68881 instruction supporting this direction of transfer. These macros request the passing of the instruction as a parameter to be consistent with the other macros. The k-factor, requested by REGMEMP, may be passed in data register D0 or as immediate data. Thus:

[k-factor] = 0
or
[k-factor] = #xxxx

A k-factor in the range +1 to +17 indicates the desired number of significant digits in the decimal mantissa after conversion to packed decimal format. A k-factor in the range -64 to 0 indicates the desired number of significant digits to the right of the decimal point in a fixed-point format.
The general instruction macro which performs "floating-point register to floating-point register" operations is:

```
REGREG Function,FPn,FPm,[FPq]
```

Since the MC68881 performs all data manipulations in extended precision (no user-specified precisions), only one macro is needed to support these general instructions. The new parameter introduced, [FPq], supports the one special case general instruction "FSINICOS" which generates the sine placing it in FPM and generates the cosine and placing it in FPq.

The constants supported on-chip by the MC68881 are available to the programmer from the coprocessor ROM with the macro:

```
FMOVEROM #CC,FPn
```

CC is the hex number representing the constant to be accessed. A list of the constants and their corresponding identification numbers is found on lines 660-681 of the EQUATE table in the demonstration software found in APPENDIX A MACROS.

The two macros which move data into or out of the coprocessor control, status, and instruction address registers are:

- MOVINCSI SourceEA,Register
- MOVOUCSI Register,DestinationEA

MOVINCSI moves data into the control, status, or instruction address registers, and MOVOUCSI moves data out. The register field is specified by CONTROL, STATUS, or IADDRESS corresponding to the register to be transferred.

As the M68000 microprocessors use the MOVEM instruction to move multiple registers into and out of memory, the MC68881 also supports moving multiple floating-point registers. The macros which support the movement of multiple floating-point data registers are:

```
FMOVEMMR SourceEA,fp0,fp1,fp2,fp3,fp4,fp5,
f6,f7,Postincrement
FMOVEMRM fp0,fp1,fp2,fp3,fp4,fp5,
f6,f7,DestinationEA,Preincrement
```

Each parameter of the fp0, fp1,...,fp7 string represents the selection bit for that floating-point register. If a register is to be moved, then the corresponding parameter in the macro call must be set to a one (otherwise, the bit must be set to a zero). If the programmer decides to use the indirect addressing with postincrement mode in the FMOVEMMR macro, then the postincrement field must be set to a Y; otherwise, this parameter must be set to a N. The same system applies to the FMOVEMRM macro with respect to the predecrement field (Y if addressing mode is used, N if not).

The no operation or synchronizing instruction is supported by the macro:

```
FNOPP
```

No parameters are necessary to perform this function. Only one macro is needed to support the branch instruction class, the conditional branch:

```
FBCC.[Size] Condition,Label
```

The size specification allows the macro to distinguish between a long branch and a short branch. If the size is not specified, the default is long. The user must specify the condition to be tested in the condition field. A list of the conditions and their corresponding label is found on lines 67-98 of the EQUATE table found in APPENDIX A MACROS. If after execution, the condition is satisfied, the macro will cause a branch to label.

The two macros which support the MC68881 decrement and branch and the conditional set instructions are:

```
FBSECC Condition,On,Label
FSESC Condition,Label
```

In both macros, the condition and label parameters serve the same purpose as those of the FBCC macro. The conditions are listed on lines 67-98 of the EQUATE table found in APPENDIX A MACROS. Note that On of FSECC cannot be equal to DO because it is used as a work register by the macro. Any other data register may be used.

The save or restore of the internal or visible state of the coprocessor is executed by the macros:

```
FSAVEST - (An)
FRESTRST (An) +
```

To ensure proper restoration of the MC68881 state, the FRESTRST command should be used only on data stacked by the FSAVEST command. One exception to this rule is for a software reset. A software reset of the MC68881 occurs if a null save format word is stored on the stack and then restored into the MC68881. This can be used by the operating system to initialize the MC68881 when starting a new task.

### Theory of Operation

The following paragraphs provide information on how to develop macros for the user who will either create his own, or need to modify the demonstration software to suit a particular application. The coprocessor recognizes each coprocessor instruction by the specific bit pattern written to the various coprocessor registers. The save instruction is an exception because it is initiated with a read from the save interface register. For each coprocessor instruction type class, a unique format for the bit pattern exists which is the basis for instruction grouping into macros. A detailed description of the macro development as well as a general discussion of each type instruction class follows.

All macros are developed following a simple protocol:
1. Know the bit pattern of the information to be written to the coprocessor (the instruction to be performed) and write it to the appropriate interface register, 2. Test for the known possible responses from the coprocessor, 3. Perform requested operation (if any), and 4. Test for the release of the main processor. If it is necessary to add exception detection, the programmer must add software to compare the response to the appropriate pre-exception or post-exception bit pattern and call a user-specified exception processing macro or subroutine if the result is positive.

Each MC68881 instruction follows a specific protocol starting from the write of the operation until the receipt of the null primitive. MEMREGn (n=B, W, L, S, D, X, P) and MOVINCSI follow the sequence shown in Figure 11. REGMEMn (n=B, W, L, S, D, X, P) and MOVOUCSI follow the sequence shown in Figure 12. Instructions represented by these REGMEMn macros generate at least one null come-again response after the initial write to the
command interface register to allow the coprocessor to perform the specified operation before the transfer of data to memory. FMOVEMMR and FMOVEMRM follow the protocols seen in Figures 13 and 14, respectively. The demonstration software only supports the static forms of the instruction, i.e., the bit mask is transferred in the instruction rather than in a main processor data register (as in the dynamic form). Thus, Figures 13 and 14 represent only the static forms of the move multiple floating-point register instructions. The REGREG macro follows the sequence of Figure 15. The FDBCC, FBCC, and FSCC macros basically execute the same code except for the function to be performed after evaluating the result of the conditional test as seen in Figure 16. Figures 17 and 18 represent the protocol followed by the FSAVE and FRESTORE instruction, respectively.

The macro detail will be explained by discussing an example of a general coprocessor instruction, the MEMREGW (lines 435–470) macro. This macro performs a floating-point operation on a word datum at an effective address pointed to by source EA address, and leaves the result in a specified floating-point register. The macro call takes the form:

MEMREGW Function, SourceEA, FPn

The function, SourceEA, and FPn are all parameters passed to the macro. Function is the operation of the MC6881 instruction, and FPn (n equals 0–7) is the specific floating-point register used. Both of these parameters represent a binary bit pattern. Thus, the need for an EQUATE table arises. An “EQU” statement (refer to the M68000 Assembler Manual) defines a symbol as a binary value when referenced anywhere in the source code. In the EQUATE table (lines 25–61) found in APPENDIX A MACROS, all general floating-point instructions are assigned the appropriate bit pattern (e.g., FMOVE EQU $00) to represent the extension field of the command word in Figure 8. Also, as seen on lines 119–126, each floating-point register (FPn) is equated to its corresponding numerical value (e.g., FP6 EQU $06). The other parameter passed to the MEMREGW macro is the effective address (SourceEA).
Since parameters are separated by commas in the format of these macro calls, the indexed register indirect with offset addressing mode, (d[An,Dn]), cannot be passed as a single parameter. The comma between An and Dn causes the assembler to see this effective address as two parameters; therefore, the macro will be passed one additional parameter for this case. Anticipating this case, when four parameters are passed in, the macro simply recombines the appropriate two parameters and reconstructs the effective address as a single parameter. The first line of this macro (line 452) is a conditional assembly command where the assembler tests for the occurrence of a fourth parameter, signifying the use of the indexed address mode. This test is done by comparing this parameter ("4") against a null character string (""), if a fourth parameter is present, a separate routine (lines 461-469) will be used to combine parameters 2 and 3 to reconstruct the effective address parameter.

Once the assembler has chosen which routine to assemble, the next task entails developing the command word shown on line 453 of the listing:

```
MOVEM.W
#$5000+(3<<7)+A1,MC68881+COMMAND
```

This task demonstrates how the command word is formed for all the effective addressing modes except the
indexed modes. The assembler, instructed by the arithmetic operator $+$, adds the three fields to generate the proper command word. The immediate data is the command word developed by the addition of the isolated fields seen in Figure 8. The command word base, 5000, represents the op-class 2 and data format for a word operand (RX). Each macro for a general instruction will have a unique command word base (shown in Table 1) specifying the op-class and data precision. A good understanding of the command word structure in Table 1 is helpful in developing general instruction macros.

The second field, Ry in Figure 8, is added to the command word by the assembler and represents the number of the floating-point register used in the transfer. This parameter is passed to the macro by the programmer as the third parameter, FPn. The symbol, $<<$, causes the assembler to shift the value of the third parameter to the left seven bits, placing it in its proper position in the command word.

The third field of the summation is the extension field which specifies the binary representation of the instruction to be performed. These representations are shown in the demonstration software in lines 25-61.

Note, when the addressing mode is indexed, the extension field remains the first parameter passed to the macro, but the Ry field becomes the fourth parameter.
occur because exceptions are not allowed. By only testing the response register (line 454) for the null come-again, the main processor will pass the data when it reads any response other than the null come-again response.

This macro, as well as all other macros except FSA-VEST and FRESTRST, must test the response register for the coprocessor release of the main processor. This service protects against spurious protocol violations. Protocol violations are unexpected accesses to the MC68881 interface registers. For example, the coprocessor may be expecting data to be written to the operand register but instead receives a write to the command register. A spurious violation occurs when an expected register access occurs sooner than expected in systems where the processor and coprocessors are running at different clock speeds. Since exceptions are assumed to be disabled by the macros, the CA bit is monitored to determine the coprocessor state. When CA is set to zero, the main processor is released. The following instructions perform this function throughout the macro definitions:

\[ \text{NULREL} \quad \text{TST:B} \quad \text{MC68881 + RESPONSE} \quad \text{BMI.S} \quad \text{NULREL} \]

In summary, this example defines the sequence to be executed in all macros of the general instructions op-class 011. Each macro causes the main processor to write the move-in operation to the command register and to read the response register until asked to pass the data. After evaluating the effective address of the data and writing it to the operand register, the main processor rereads the response register until released by the coprocessor.

The packed BCD, double precision, and extended precision operations would require the use of several other conditional assembly instructions to support all the addressing modes that the byte, word, long word, and single precision macros allow. These instructions are necessary due to the fact that multiple accesses from memory are required to transfer data through the 32-bit operand register. To simplify this application, these three precision, MEMREGD, MEMREGX, and MEMREGP, are only supported by the address register indirect addressing mode. The other addressing modes can be implemented by following the demonstration software as an example.

The move-out macros (REGMEMn, n=R, W, L, S, D, X, P) of op-class 011 of the general instruction class are structured in the same manner as MEMREGn (n=R, W, L, S, D, X, P). Coprocessor distinction between the move-in and the move-out operations result from the different op-class specifications within the command word.

The one difference between the two op-classes is in the packed BCD macros. This difference is due to the nature of the MC68881 FMOVE out packed BCD from the coprocessor instruction which requires the user to submit additional information to the coprocessor: the k-factor. The k-factor is passed to the operand register from either a data register or as immediate data in the command word. To be able to handle all data registers, the packed BCD macro would be extensive using elaborate conditional statements. Therefore, the programmer is only allowed to use data register D0, which fixes that part of
Figure 19, F-Line Emulation Sequence
the extension field representing the data register as a constant. Thus, only one pair of conditional instructions is needed.

The REGREG macro supports op-class 000 which performs a coprocessor register-to-register operation. No services are needed of the main processor other than to submit the coprocessor instruction. Therefore, after writing the command word to the command register, the response register is queried until the null primitive is granted. The several other conditional assembly statements in REGREG support the unique general arithmetic instruction, FSINCS. Since this instruction requires two destination floating-point registers for the results of the operation (FPm and FPq), another parameter must be passed to the coprocessor. The conditional assembly statement tests for the existence of a fourth parameter. To be able to support this instruction in the REGMEMn and MEMREGn macros, similar procedures should be followed. An example is implemented in the MEMREGB macro (line 372-434).

Op-class 010 with Rx equal to 111 represents the operation performing the access of the coprocessor constants (FMOVEROM). A command word specifying the constant to be retrieved is written to the command register. Since no further services of the main processor are needed, the remaining function is to test the response register for the release signal (CA equals zero).

Both macros, MOVINCSI (op-class 100) and MOVOUNCSI (op-class 101) move the coprocessor system registers. Each performs the same instruction sequence as MEMREGn and REGMEMn, respectively, with the only difference being the value of the command word. The move multiple coprocessor system register instructions are not supported by the macros.

The final op-classes of the general instructions to be discussed are those corresponding to the movement of multiple floating-point data registers (FMOVEMRM and FMOVEMMR). Due to the nature of the M68000 Family memory organization, the macros are constructed differently. The user specifies which registers are moved by selecting the corresponding parameter in the macro call, and the coprocessor detects which registers are affected by the bit mask specified in the command word. The binary bit mask is formed by the parameter list. The list is treated as a string of concatenated bits which is required by the MC68881 to represent the register select mask.

Since the floating-point data registers are 96 bits wide, three consecutive accesses of 32 bits each must be made to acquire the data. FMOVEMRM (lines 761-834) organizes this data so that the high-order bit is situated in the low-order memory. The coprocessor delivers FP0 first (if selected) and FP7 last, except when the indirect addressing with predecrement mode is being used. In which case, the coprocessor sends FP7 first and FP0 last so that FP0 is always placed in low memory. Therefore, the conditional test for the predecrement mode is required to reverse the order of the bit mask sent to the coprocessor. The FMOVEMMR (lines 835-881) macro moves data into the registers by moving FP0 first as the coprocessor always expects FP0 first. The FMOVEMMR macro does not allow the predecrement addressing mode.

The second type class to be discussed is the branch instruction class which is supported by the FBCC macro. The main processor writes the conditional predicate (CPRED) to the condition register and reads the response register until signaled to be released. Then the T/F bit of the response primitive is examined, and if status indicates, the branch is taken.

The FDBCC and FSBC macros support the conditional type instructions. Both macros follow the same protocol as FBCC (i.e., write CPRED to the condition register, read the response register, and after being released, perform requested function if condition satisfied). All of the branch and conditional macros must modify a data register which serves as a temporary variable. When the coprocessor grants the null release primitive, the T/F bit is also passed in the response. As the MC68881 does not expect another response register access, the response is saved in D0 so the CA bit can be tested. When CA equals zero, the T/F bit is already available in D0.

The no-operation macro exists for the no-operation or synchronizing FNOOP instruction. It is a branch never instruction. The main processor writes the second word of the coprocessor instruction to the condition register and queries the response register until released by the coprocessor.

One MC68881 conditional instruction not implemented is the conditional trap (FTRAPcc) instruction because the MC68020 has these coprocessor traps. The trap instruction is not available on the MC68000, the MC68008, or the MC68010. To cause a trap from the user space in a MC68000/MC68008/MC68010 system, the overflow bit in the control register can be set, and the TRAPV instruction executed. However, the trap handler can not distinguish between the simulated coprocessor condition and the overflow condition that would normally use this trap vector.

The final two coprocessor instruction types to be discussed are the save and restore performed by macros FSASVEST and FRESTRST, respectively. Only one addressing mode is supported in the macros. Several other conditional assembly instructions, similar to those in the FMOVEMRM and FMOVEMMR macro, can be implemented to utilize more addressing modes. To initiate the save sequence, the main processor reads the format word from the save register. This 16-bit register is reread until the high-order byte no longer contains a 01 (coprocessor busy). At this point, the length (in bytes) of the coprocessor data to be transferred resides in the low-order byte of the format word. The main processor isolates this length and begins to transfer the data from the operand register (making long word accesses) to memory via the indirect addressing with predecrement addressing mode. After saving the invisible portion of the coprocessor state, the main processor stores the format word at the top of the stack, in low-order memory. This assures proper restoration of the MC68881 state when the FRESTRST macro is executed. In FRESTRST, the main processor writes the previously saved format word from memory to the restore register, reads the restore register, and begins writing the stored data to the operand register until the proper number of bytes has been transferred. Indirect addressing with postincrement addressing mode is used.
In summary, the performance of the MC68881, driven as a peripheral in a MC68000/MC68008/MC68010 system, is enhanced by using the macro approach. This is primarily due to the fact that most of the instruction decode is done at assembly. This in-line code is upwardly source code compatible to a MC68020 system via re-compilation or reassembly. For instance, the following code provides an example of how to alter a macro (for reassembly) in order to acquire floating-point source code compatibility when porting the user software to an MC68020 system (equate table must be deleted):

MEMREGB MACRO
"1.B" "2,3"
ENDM

The macro call will remain the same. For example, this macro call:

MEMREGB FADD,D0,FPO

expands to create the following MC68881 floating-point ADD source code when used in conjunction with the previous macro definition:

FADD D0,FPO

A few consequences of this technique exist: 1) the object code is not MC68881 replaceable because if the code were moved up to a MC68020/MC68010 system, the MC68881 would still be a peripheral processor in user data space (to benefit from the MC68020 coprocessor interface, the macros would have to be changed and the user program reassembled), 2) a macro library and/or other routines are required to contain the macro software, 3) the full environment is not presented to the user as not all addressing modes nor the FTRAPcc instruction are supported, all checking done by the MC68020 is not implemented (e.g., illegal format errors), and exceptions are not enabled, 4) the MC68881 is not an independent operating hardware device because peripheral I/O access is used, and 5) the demonstration software does not support the M68000 immediate addressing mode.

F-LINE TRAP SOFTWARE EMULATION

As an alternative to using macros or in-line code, an F-line trap emulation could be implemented in an MC68000/MC68008/MC68010 system when the user requires the user program object code containing MC68881 instructions to be upwardly compatible to the MC68020 without recompiling, reassembling, or relinking. By using this approach, the coprocessor will be driven as a peripheral from supervisor space by supervisor software. Complete source and object code compatibility with the MC68881 instruction set can be maintained.

Because some M68000 systems separate user and supervisor space, different types of emulations must be developed. This application note includes two examples of F-line emulation of the general instruction operation performed on data moved into the coprocessor: the protected and unprotected versions. The software for other types such as move outs can be inferred from the examples given. The protected version (APPENDIX B PROTECTED F-LINE EMULATION SOFTWARE) is used on systems which segregate user and system address spaces. The unprotected version (APPENDIX C UNPROTECTED F-LINE EMULATION SOFTWARE) can be used on any M68000 system which allows direct access to user spaces from the supervisor state.

Functional Description

If the coprocessor instruction were decoded by the trap routine to determine the addressing modes used to access the instruction operands, then a significant overhead would be incurred with a commensurate loss of performance. Hence, the demonstration software presumes a single addressing mode will always be used. This is register indirect, (A0). If the programmer desires to use other addressing modes, this can be accomplished by simply performing:

LEA EA,A0

before executing the floating-point instruction. Note, the LEA instruction will not work when using PC relative addressing mode in a system that splits program and data spaces (although this is rarely encountered). Also, as implemented in the macro approach, no error or exception checking is performed by the F-line emulator approach. In the examples, all memory to floating-point register operations are supported including FMOVECR and all FPn operations.

Theory of Operation

The following paragraphs provide information for users creating their own F-line trap emulations. The read-write protocol of the move-in macros (Figure 11) is implemented in both the protected and unprotected forms of the emulation. The F-line trap emulation differs from the protocol of the macro approach. In the emulation, after the main processor has transferred the data to the coprocessor, the final read of the response register is no longer needed. Sufficient time will expire between any two consecutive floating-point instructions due to the overhead of the F-line trap which ensures that no spurious protocol violations will occur.

A flowchart of the unprotected emulation version is seen in Figure 19. In the protected version, the same sequence of events occurs with the exception that the floating-point instruction and source operands are accessed in user memory from supervisor space utilizing the M68010 MOVES and MOVEC instructions. The unprotected version (APPENDIX C UNPROTECTED F-LINE EMULATION SOFTWARE) is referenced.

When a coprocessor instruction is encountered, the F-line trap is taken. The location of the coprocessor instruction (program counter) and other information (depending on the main processor executing the instruction) is placed on the stack. The data at the program counter location (the operation word of the coprocessor instruction) is examined to determine whether the instruction is a general type (line 25). If so, the second word (the command word) is written to the command register (line 30). Then, the main processor queries the response register until the coprocessor no longer processes the previous instruction (no null come-again).
Next, the op-class specified in the command word is examined to determine the main processor's next action. First, the main processor tests for the move multiple coprocessor registers (data or system) into or out of the MC68881 (op-classes with high-order bits set, e.g. 1xx) in lines 33-34. If found, the main processor would jump to a routine to handle this special function. This function is not implemented in this application but is a straightforward routine.

Subsequently, testing for a floating-point register-to-register operation occurs in line 35. In this case, no further services are needed of the main processor, and a jump to the RTE instruction is taken.

Finally, a distinction between the move-in and move-out operations is made (line 37-38). (An additional routine can be developed to support the move-out sequence.) When a move-in operation has been identified, the main processor then extracts the precision of the external operand from the command word. If the instruction is found to be a FMOVECR (precision 111), the main processor immediately branches to the RTE instruction. Otherwise, the main processor branches to the small routines for handling the respective data transfers. Since long words, packed BCD, single, double, and extended precision data transfers all require at least one 32-bit data transfer, one routine handles all five data types (lines 34-56). Two other routines (lines 59-60 and lines 63-64) support the byte and word transfers. After the data has been delivered to the coprocessor, the main processor returns from exception via the RTE instruction (the instruction which completes the F-line trap and re-enters the user program).

In both versions of the F-line trap emulation, the work registers were stored at the beginning of the routine and then restored prior to the exit.

In summary, the F-line emulation trap is initiated when the main processor identifies a coprocessor instruction by a hexadecimal F in the most-significant nibble of the first word of the instruction (and takes the F-line trap).

The object code containing the MC68881 instruction is upward compatible to an MC68020 system without re-compiling, reassembling, or relinking.

The major consequence in implementing an F-line emulation instead of in-line code is the time factor incurred by both the overhead of the F-line trap and the instruction decode in the trap routine. Listed in Table 2 are the clock cycles required to perform the various operations using the two F-line emulations and the macro approaches. Even with the overhead associated with the F-line emulation this approach offers a speed advantage over floating-point software packages and at the same time maintains MC68020/MC68881 upward compatibility. Timings are based on a no-wait state system (four clock cycle bus cycle). The MC68881 overhead is not taken into account because the main processor is released to perform the next instruction of the user program while the coprocessor executes its instructions. FMOVE is an example instruction that could be replaced by any of the general instructions (excluding the move multiples).

**CONCLUSION**

The MC68881 floating-point coprocessor can be utilized as a peripheral in a MC68000/MC68030/MC68010 system by either directly driving the device as a peripheral or by simulating the complete coprocessor instruction set. Either method, depending on the application, is sufficient to utilize the high performance of the MC68881 and offers superior speed and versatility over floating-point software packages. The macros or in-line code of this application provide a faster way to access the device for the users interested in achieving the highest performance of the MC68881. Alternately, for applications that can trade-off performance to achieve object code upward compatibility with MC68020 systems, an example of an F-line emulation trap has also been included.

**Table 2. Operation Execution Time (Clock Cycle) for MC68010**

<table>
<thead>
<tr>
<th>Operation (MC68881)</th>
<th>Macro</th>
<th>F-Line (Protected)</th>
<th>F-Line (Unprotected)</th>
</tr>
</thead>
<tbody>
<tr>
<td>F(op),B EA,FPhn</td>
<td>88</td>
<td>410</td>
<td>342</td>
</tr>
<tr>
<td>F(op),W EA,FPhn</td>
<td>88</td>
<td>410</td>
<td>342</td>
</tr>
<tr>
<td>F(op),L EA,FPhn</td>
<td>96</td>
<td>462</td>
<td>398</td>
</tr>
<tr>
<td>F(op),S EA,FPhn</td>
<td>96</td>
<td>462</td>
<td>398</td>
</tr>
<tr>
<td>F(op),D EA,FPhn</td>
<td>124</td>
<td>516</td>
<td>436</td>
</tr>
<tr>
<td>F(op),X EA,FPhn</td>
<td>156</td>
<td>570</td>
<td>474</td>
</tr>
<tr>
<td>F(op),P EA,FPhn</td>
<td>156</td>
<td>570</td>
<td>474</td>
</tr>
<tr>
<td>F(op),X FPhn,FPhn</td>
<td>40</td>
<td>296</td>
<td>236</td>
</tr>
<tr>
<td>FMOVECR #ccc,FPhn</td>
<td>40</td>
<td>370</td>
<td>316</td>
</tr>
</tbody>
</table>

(op) = MOVE, ADD, SUB
APPENDIX A
MACROS

******************************************************************************
* SOURCE CODE TO DRIVE THE MC68881 AS A PERIPHERAL                      *
* TO NOT SHOW MACRO EXPANSION IN THE LIST FILE DELETE 'OPT MEX' BEFORE    *
* ASSEMBLY.                                                               *
* TO SHOW THE CONDITIONAL ASSEMBLY INSTRUCTIONS IN THE MACRO EXPANSION   *
* DELETE THE 'OPT NOCL' (LOCATED AFTER THIS BOX) BEFORE ASSEMBLY.          *
******************************************************************************

OPT NOCL

OPT MEX

******************************************************************************
* THIS IS THE EQUATE FILE TO SUPPORT THE MACROS USED                      *
* TO DRIVE THE MC68881 AS A PERIPHERAL                                   *
* WITH THE 68000 FAMILY                                                   *
******************************************************************************

THESE ARE THE INSTRUCTION BIT PATTERN EQUIVALENTS

FMOVE EQU $00 MOVE
FINT EQU $01 INTERGER PART
FSINH EQU $02 SINH
FSQRT EQU $04 SQUARE ROOT
FLOGNP1 EQU $06 LOGN (1/x)
FETOXM1 EQU $08 [(E**X)-1]
FTANH EQU $09 TANH
FATAN EQU $0A ARCTAN
FASIN EQU $0C ARCSIN
FATANH EQU $0D ARCTANH
FSIN EQU $0E SINE
FTAN EQU $0F TANGENT
FETO EQU $10 E**X
FIMOTOX EQU $11 2**X
FTENTOX EQU $12 10**X
FLOGN EQU $14 LOGN
FLOG10 EQU $15 LOG10
FLOG2 EQU $16 LOG2
FABS EQU $18 ABSOLUTE VALUE
FCOSH EQU $19 COSH
FSIN EQU $1A NEGATE
FSINH EQU $1B ARCCOS
FSINH EQU $1D COSINH
FGETEXP EQU $1E GET EXPONENT
FGETMAN EQU $1F GET MANTISSA
FDIV EQU $20 DIVIDE
FMCD EQU $21 MODULO REMAINDER
FADD EQU $22 ADD
FMUL EQU $23 MULTIPLY
FSGLDIV EQU $24 SINGLE DIVIDE
FREM EQU $25 IEEE REMAINDER
FSCALE EQU $26 SCALE EXPONENT
FSGLMUL EQU $27 SINGLE MULTIPLY
FSUB EQU $28 SUBTRACT
FCMP EQU $38 COMPARE
FTST EQU $3A TEST
FSINCOS EQU $30 SIMULTANEOUS FP SINE AND COSINE
<table>
<thead>
<tr>
<th>Line</th>
<th>Mnemonic</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>EQ</td>
<td>$01</td>
<td>EQUAL</td>
</tr>
<tr>
<td>64</td>
<td>NEQ</td>
<td>$0E</td>
<td>NOT EQUAL</td>
</tr>
<tr>
<td>65</td>
<td>GT</td>
<td>$12</td>
<td>GREATER THAN</td>
</tr>
<tr>
<td>66</td>
<td>LT</td>
<td>$14</td>
<td>LESS THAN</td>
</tr>
<tr>
<td>67</td>
<td>NEQ</td>
<td>$0E</td>
<td>NOT LESS THAN</td>
</tr>
<tr>
<td>68</td>
<td>GT</td>
<td>$12</td>
<td>GREATER OR LESS THAN</td>
</tr>
<tr>
<td>69</td>
<td>NEQ</td>
<td>$0E</td>
<td>LESS THAN OR EQUAL</td>
</tr>
<tr>
<td>70</td>
<td>GT</td>
<td>$12</td>
<td>GREATER THAN OR EQUAL</td>
</tr>
<tr>
<td>71</td>
<td>EQ</td>
<td>$01</td>
<td>NOT (LESS THAN OR EQUAL)</td>
</tr>
<tr>
<td>72</td>
<td>NEQ</td>
<td>$0E</td>
<td>GREATER THAN OR LESS THAN OR EQUAL</td>
</tr>
<tr>
<td>73</td>
<td>GT</td>
<td>$12</td>
<td>GREATER OR LESS THAN OR EQUAL</td>
</tr>
<tr>
<td>74</td>
<td>LT</td>
<td>$14</td>
<td>GREATER OR LESS THAN OR LESS THAN</td>
</tr>
<tr>
<td>75</td>
<td>NEQ</td>
<td>$0E</td>
<td>LESS THAN OR LESS THAN OR LESS THAN</td>
</tr>
<tr>
<td>76</td>
<td>GT</td>
<td>$12</td>
<td>GREATER THAN OR LESS THAN OR LESS THAN</td>
</tr>
<tr>
<td>77</td>
<td>NEQ</td>
<td>$0E</td>
<td>LESS THAN OR LESS THAN OR LESS THAN</td>
</tr>
<tr>
<td>78</td>
<td>GT</td>
<td>$12</td>
<td>GREATER THAN OR LESS THAN OR LESS THAN</td>
</tr>
<tr>
<td>79</td>
<td>LT</td>
<td>$14</td>
<td>GREATER OR LESS THAN OR LESS THAN</td>
</tr>
<tr>
<td>80</td>
<td>NEQ</td>
<td>$0E</td>
<td>LESS THAN OR LESS THAN OR LESS THAN</td>
</tr>
<tr>
<td>81</td>
<td>GT</td>
<td>$12</td>
<td>GREATER THAN OR LESS THAN OR LESS THAN</td>
</tr>
<tr>
<td>82</td>
<td>NEQ</td>
<td>$0E</td>
<td>LESS THAN OR LESS THAN OR LESS THAN</td>
</tr>
<tr>
<td>83</td>
<td>GT</td>
<td>$12</td>
<td>GREATER THAN OR LESS THAN OR LESS THAN</td>
</tr>
<tr>
<td>84</td>
<td>LT</td>
<td>$14</td>
<td>GREATER OR LESS THAN OR LESS THAN</td>
</tr>
<tr>
<td>85</td>
<td>NEQ</td>
<td>$0E</td>
<td>LESS THAN OR LESS THAN OR LESS THAN</td>
</tr>
<tr>
<td>86</td>
<td>GT</td>
<td>$12</td>
<td>GREATER THAN OR LESS THAN OR LESS THAN</td>
</tr>
<tr>
<td>87</td>
<td>LT</td>
<td>$14</td>
<td>GREATER OR LESS THAN OR LESS THAN</td>
</tr>
<tr>
<td>88</td>
<td>NEQ</td>
<td>$0E</td>
<td>LESS THAN OR LESS THAN OR LESS THAN</td>
</tr>
<tr>
<td>89</td>
<td>GT</td>
<td>$12</td>
<td>GREATER THAN OR LESS THAN OR LESS THAN</td>
</tr>
<tr>
<td>90</td>
<td>LT</td>
<td>$14</td>
<td>GREATER OR LESS THAN OR LESS THAN</td>
</tr>
<tr>
<td>91</td>
<td>NEQ</td>
<td>$0E</td>
<td>LESS THAN OR LESS THAN OR LESS THAN</td>
</tr>
<tr>
<td>92</td>
<td>GT</td>
<td>$12</td>
<td>GREATER THAN OR LESS THAN OR LESS THAN</td>
</tr>
<tr>
<td>93</td>
<td>LT</td>
<td>$14</td>
<td>GREATER OR LESS THAN OR LESS THAN</td>
</tr>
<tr>
<td>94</td>
<td>NEQ</td>
<td>$0E</td>
<td>LESS THAN OR LESS THAN OR LESS THAN</td>
</tr>
<tr>
<td>95</td>
<td>GT</td>
<td>$12</td>
<td>GREATER THAN OR LESS THAN OR LESS THAN</td>
</tr>
<tr>
<td>96</td>
<td>LT</td>
<td>$14</td>
<td>GREATER OR LESS THAN OR LESS THAN</td>
</tr>
<tr>
<td>97</td>
<td>NEQ</td>
<td>$0E</td>
<td>LESS THAN OR LESS THAN OR LESS THAN</td>
</tr>
<tr>
<td>98</td>
<td>GT</td>
<td>$12</td>
<td>GREATER THAN OR LESS THAN OR LESS THAN</td>
</tr>
<tr>
<td>99</td>
<td>LT</td>
<td>$14</td>
<td>GREATER OR LESS THAN OR LESS THAN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line</th>
<th>Mnemonic</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>COMMAND</td>
<td>$0A</td>
<td>COMMAND REGISTER</td>
</tr>
<tr>
<td>101</td>
<td>RESPONSE</td>
<td>$00</td>
<td>RESPONSE REGISTER</td>
</tr>
<tr>
<td>102</td>
<td>OPER</td>
<td>$01</td>
<td>OPERAND REGISTER</td>
</tr>
<tr>
<td>103</td>
<td>COND</td>
<td>$02</td>
<td>CONDITION REGISTER</td>
</tr>
<tr>
<td>104</td>
<td>SAVE</td>
<td>$04</td>
<td>SAVE REGISTER</td>
</tr>
<tr>
<td>105</td>
<td>RESTORE</td>
<td>$06</td>
<td>RESTORE REGISTER</td>
</tr>
<tr>
<td>106</td>
<td>REGSEL</td>
<td>$14</td>
<td>REGISTER SELECT</td>
</tr>
<tr>
<td>107</td>
<td>CONTROL</td>
<td>$9000</td>
<td>MC68881 CONTROL REGISTER</td>
</tr>
<tr>
<td>108</td>
<td>STATUS</td>
<td>$8000</td>
<td>MC68881 STATUS REGISTER</td>
</tr>
<tr>
<td>109</td>
<td>IADDRESS</td>
<td>$8400</td>
<td>MC68881 INSTRUCTION ADDRESS REGISTER</td>
</tr>
<tr>
<td>110</td>
<td>TTBIT</td>
<td>$0</td>
<td>TRUE/FALSE BIT OF THE RESPONSE REGISTER</td>
</tr>
<tr>
<td>111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>FP0</td>
<td>$00</td>
<td>FLOATING POINT REGISTER #0</td>
</tr>
<tr>
<td>113</td>
<td>FP1</td>
<td>$01</td>
<td></td>
</tr>
<tr>
<td>114</td>
<td>FP2</td>
<td>$02</td>
<td></td>
</tr>
<tr>
<td>115</td>
<td>FP3</td>
<td>$03</td>
<td></td>
</tr>
<tr>
<td>116</td>
<td>FP4</td>
<td>$04</td>
<td>FLOATING POINT REGISTER #4</td>
</tr>
<tr>
<td>117</td>
<td>FP5</td>
<td>$05</td>
<td></td>
</tr>
<tr>
<td>118</td>
<td>FP6</td>
<td>$06</td>
<td></td>
</tr>
<tr>
<td>119</td>
<td>FP7</td>
<td>$07</td>
<td></td>
</tr>
</tbody>
</table>

MOTOROLA
22
AN947/D
MC68881 SINGLE PRECISION FP-REG. VALUE TO MEMORY OPERATION

WHERE: INSTRUCTION- FP INSTRUCTION NUOMONIC (I.E. FMOVE)  
FPM- SOURCE FP REGISTER  
<EA>- DESTINATION ADDRESSING MODE

NO REGISTERS MODIFIED OR DESTROYED!

VALID ADDRESSING MODES:  
DN, (AN)+, -(AN), D(AN), D(AN,IX)

..........................

REGEMS MACRO

IFC \"4\", IS <EA>= INDIRECT WITH INDEXING
MOVE.W #$6400+(\2<<7)+1,MC68881+COMMAND MEM. TO REG. OPERATION
\@NULCA CMPI #$8900,MC68881+RESPONSE READ RESPONSE REGISTER
\@NULCA BEQ.S \@NULCA REREAD UNTIL EVALUATE EA AND TRANSFER DATA
\@NULCA MOVE.L MC68881+OPER,\3 LOW ORDER WORD
\@NUREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE?
\@NUREL BMI.S \@NUREL BRANCH UNTIL NULL RELEASE

IFC \"4\", IS <EA> NOT = INDIRECT WITH INDEXING
MOVE.W #$6400+(\2<<7)+1,MC68881+COMMAND MEM. TO REG. OPERATION
\@NULCA CMPI #$8900,MC68881+RESPONSE READ RESPONSE REGISTER
\@NULCA BEQ.S \@NULCA REREAD UNTIL EVALUATE EA AND TRANSFER DATA
\@NULCA MOVE.L MC68881+OPER,\3 SINGLE PRECISION DATA TRANSFER
\@NUREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE?
\@NUREL BMI.S \@NUREL BRANCH UNTIL NULL RELEASE

MC68881 LONG WORD LENGTH FP-REG. VALUE TO MEMORY OPERATION

REGEMS MACRO

IFC \"4\", IS <EA>= INDIRECT WITH INDEXING
MOVE.W #$6400+(\2<<7)+1,MC68881+COMMAND MEM. TO REG. OPERATION
\@NULCA CMPI #$8900,MC68881+RESPONSE READ RESPONSE REGISTER
\@NULCA BEQ.S \@NULCA REREAD UNTIL EVALUATE EA AND TRANSFER DATA
\@NULCA MOVE.L MC68881+OPER,\3 LONG WORD TRANSFER
\@NUREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE?
\@NUREL BMI.S \@NUREL BRANCH UNTIL NULL RELEASE
\@NUREL ENDC
\@NULCA IFIC \"4\", IS <EA> NOT = INDIRECT WITH INDEXING
MOVE.W #$6400+(\2<<7)+1,MC68881+COMMAND MEM. TO REG. OPERATION
\@NULCA CMPI #$8900,MC68881+RESPONSE READ RESPONSE REGISTER
\@NULCA BEQ.S \@NULCA REREAD UNTIL EVALUATE EA AND TRANSFER DATA
\@NULCA MOVE.L MC68881+OPER,\3 MOVE.W #$6000\lf(\2<<7)\l,MC68881+CMD. TO REG. OPERATION
\@NULCA CMPI #$8900,K68881+RESPONSE IS RESPONSE REGISTER
\@NULCA BEQ. S \@NULCA MOVE.W #$6000\lf(\2<<7)\l,MC68881+CMD. TO REG. OPERATION
\@NULCA CMPI #$8900,K68881+RESPONSE IS RESPONSE REGISTER
193 * MOVE.L MC68881+OPER, \3, \4 DATA
194 \&NUIREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE?
195 BMI.S \&NUIREL BRANCH UNTIL NULL RELEASE
196 ENDC
197 ENDM
198
199***********************************************************************
200 *
201 * MC68881 WORD LENGTH FP-REG. VALUE TO MEMORY OPERATION *
202 *
203 * REGMEMB INSTRUCTION, FPM, <EA>
204 *
205 * WHERE: INSTRUCTION= FP INSTRUCTION NUMERIC (I.E. FMOVE)
206 *
207 * FPM= SOURCE FP REGISTER
208 *
209 * <EA>= DESTINATION ADDRESSING MODE
210 *
211 * NO REGISTERS MODIFIED OR DESTROYED!
212 *
213 * VALID ADDRESSING MODES: DN, (AN)+, -(AN), D(AN), D(AN,IX)
214 *
215 * XXX.W, XXX.L, (D,PC), D(PC,IX)
216 *
217******************************************************************************
218 *
219 * REGMEMB MACRO
220 *
221 IFC \"4\", \"\" IS <EA>=INDIRECT WITH INDEXING
222 MOVE.W $7000+(\2<<7)+1, MC68881+COMMAND MEM. TO REG. OPERATION
223 \&NULCA CMPF $8900, MC68881+RESPONSE READ RESPONSE REGISTER
224 BEQ.S \&NULCA REREAD UNTIL EVALUATE EA AND TRANSFER DATA
225 MOVES MC68881+OPER, \3 WORD DATA TRANSFER
226 \&NUIREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE?
227 BMI.S \&NUIREL BRANCH UNTIL NULL RELEASE
228 ENDC
229 *
230 IFC \"4\", \"\" IS <EA> NOT = INDIRECT WITH INDEXING
231 MOVE.W $7000+(\2<<7)+1, MC68881+COMMAND MEM. TO REG. OPERATION
232 \&NULCA CMPF $8900, MC68881+RESPONSE READ RESPONSE REGISTER
233 BEQ.S \&NULCA REREAD UNTIL EVALUATE EA AND TRANSFER DATA
234 MOVES MC68881+OPER, \3, \4 WORD DATA TRANSFER
235 \&NUIREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE?
236 BMI.S \&NUIREL BRANCH UNTIL NULL RELEASE
237 ENDC
238 *
239 *
240******************************************************************************
241 *
242 * MC68881 BYTE LENGTH FP-REG. VALUE TO MEMORY OPERATION *
243 *
244 * REGMEMB INSTRUCTION, FPM, <EA>
245 *
246 * WHERE: INSTRUCTION= FP INSTRUCTION NUMERIC (I.E. FMOVE)
247 *
248 * FPM= SOURCE FP REGISTER
249 *
250 * <EA>= DESTINATION ADDRESSING MODE
251 *
252 * NO REGISTERS MODIFIED OR DESTROYED!
253 *
254 * VALID ADDRESSING MODES: DN, (AN)+, -(AN), D(AN), D(AN,IX)
255 *
256 * XXX.W, XXX.L, (D,PC), D(PC,IX)
257 *
258******************************************************************************
259 *
260 * REGMEMB MACRO
261 IFC \"4\", \"\" IS <EA>=INDIRECT WITH INDEXING
262 MOVE.W $7800+(\2<<7)+1, MC68881+COMMAND MEM. TO REG. OPERATION
263 \&NULCA CMPF $8900, MC68881+RESPONSE READ RESPONSE REGISTER
264 BEQ.S \&NULCA REREAD UNTIL EVALUATE EA AND TRANSFER
259 * MOVE.B MC68881+OPER,\3
260 \&NULLREL TST.B MC68881+RESPONSE
261 BML.S \&NULLREL
262 ENDC
263 IFNC \('4'," "); IS <EA> NOT = INDIRECT WITH INDEXING
264 MOVE.W \#8000+(\2<<7)+1,MC68881+COMMAND MEM. TO REG. OPERATION
265 \&NULLCA CMPI \#8900,MC68881+RESPONSE
266 BEQ.S \&NULLCA
267 MOVE.B MC68881+OPER,\3,\4
268 \&NULLREL TST.B MC68881+RESPONSE
269 BML.S \&NULLREL
270 ENDC
271 END

******************************************************************************

275 * MC68881 DOUBLE PRECISION FP-REG. VALUE TO MEMORY OPERATION
276 * REGEMD MACRO
277 MOVE.W \#8000+(\2<<7)+1,MC68881+COMMAND MEM. TO REG. OPERATION
278 \&NULLCA CMPI \#8900,MC68881+RESPONSE
279 BEQ.S \&NULLCA
280 MOVE.L MC68881+OPER,\3
281 MOVE.L MC68881+OPER,\4
282 \&NULLREL TST.B MC68881+RESPONSE
283 BML.S \&NULLREL
284 ENDM

******************************************************************************

289 * MC68881 EXTENDED PRECISION FP-REG. VALUE TO MEMORY OPERATION
290 * REGEMX MACRO
291 MOVE.W \#6800+(\2<<7)+1,MC68881+COMMAND MEM. TO REG. OPERATION
292 \&NULLCA CMPI \#8900,MC68881+RESPONSE
293 BEQ.S \&NULLCA
294 MOVE.L MC68881+OPER,\3
295 MOVE.L MC68881+OPER,\4
296 \&NULLREL TST.B MC68881+RESPONSE
297 BML.S \&NULLREL
298 ENDM

******************************************************************************

299 * ___________________________________________________________
300 *
301 * NO REGISTERS MODIFIED OR DESTROYED!
302 *
303 * VALID ADDRESSING MODES: (AN)
304 *
305 *
306 *
307 *
308 *
309 *
310 *
311 *
312 *
313 *
314 *
315 *
316 *
317 *
318 *
319 *
320 *
321 *
322 *
323 *
324 *

********************************************************************************
MOVE.L MC68881+OPER,8\3
\$NULREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE?
B.MI.S \$NULREL BRANCH UNTIL NULL RELEASE
ENDM

*****************************************************************************

* * * * * MC68881 PACKED BCD FP-REG. VALUE TO MEMORY OPERATION * * *
* * * * * INSTRUCTION,FPM,\<EA>, \[K-FACTOR] * * *
* * * * * WHERE: INSTRUCTION= FP INSTRUCTION MNEMONIC (I.E. FADD) * * *
* * * * * FPM= SOURCE FP REGISTER * * *
* * * * * \<EA>= DESTINATION ADDRESSING MODE * * *
* * * * * \[K-FACTOR]= OPTIONAL IMMEDIATE K-FACTOR * * *
* * * * * ***IF \[K-FACTOR] OPTION NOT TAKEN, THE K-FACTOR MUST BE PLACED IN DO! * * *
* * * * * VALID ADDRESSING MODES: (AN) * *
* * * * *
*****************************************************************************

REGMEMP MACRO
IFC \"4\", \" IS K-FACTOR IN REGISTER?
MOVE.W \#7C00+(\iff<7)+\iff,MC68881+COMMAND MEM. TO REG. OPERATION
\$NULCA CMP.I \#8900,MC68881+RESPONSE READ RESPONSE REGISTER
BEQ.S \$NULCA REREAD UNTIL TRANSFER MAIN PROCESSOR REG
MOVE.L MC68881+OPER,\3 PASS K-FACTOR FROM DO
AGAIN CMP.I \#8900,MC68881+RESPONSE READ RESPONSE REGISTER
BEQ.S \ AGAIN REREAD UNTIL EVALUATE EFFECTIVE ADDRESS
*
MOVE.L MC68881+OPER,\3 LOW ORDER LONG WORD
MOVE.L MC68881+OPER,\4 MID-ORDER LONG WORD
MOVE.L MC68881+OPER,\8 HIGH ORDER LONG WORD
\$NULREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE?
B.MI.S \$NULREL BRANCH UNTIL NULL RELEASE
ENDC
IFNC \"4\", \" IS K-FACTOR IN INSTRUCTION?
MOVE.W \#6C00+(\iff<7)+\iff,MC68881+COMMAND MEM. TO REG. OPERATION
\$NULCA CMP.I \#8900,MC68881+RESPONSE READ RESPONSE REGISTER
BEQ.S \$NULCA REREAD UNTIL EVALUATE EFFECTIVE ADDRESS
*
MOVE.L MC68881+OPER,\3 LOW ORDER WORD
MOVE.L MC68881+OPER,\4 MID-ORDER WORD
MOVE.L MC68881+OPER,\8 HIGH ORDER WORD
\$NULREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE?
B.MI.S \$NULREL BRANCH UNTIL NULL RELEASE
ENDC
ENDM

*****************************************************************************

* * * * * MC68881 BYTE IN MEMORY OR IN Dn TO FP-REG. OPERATION * * *
* * * * * MEMREGB INSTRUCTION,\<EA>,FPM * *
* * * * * WHERE: INSTRUCTION= FP INSTRUCTION MNEMONIC (I.E. FADD) * *
* * * * * \<EA>= SOURCE ADDRESSING MODE * *
* * * * * FPM= DESTINATION REGISTER * *
* * * * * NO REGISTERS MODIFIED OR DESTROYED! * *
* * * * * VALID ADDRESSING MODES: DN, (AN)+, -\(AN), D(AN), D(AN,IX) * *
* * * * * \(XX.X, \(XX.X, L, (D,FC), D(FC,IX) * *
* * * * * THE COMMENTED OUT CODE SHOWS HOW A USER MAY IMPLEMENT FSINCONCOS * *
* * * * * IN A MEM. TO REG. TRANSFER USING THE FOLLOWING INSTRUCTION FORMAT: * *
* * * * * MEMREGB INSTRUCTION,\<EA>,FPM,FPPQ (FPPQ= 2ND DESTINATION REG.) * *

*****************************************************************************

MOTOROLA
26
AN847/D
******************************************************************************
~GB MACRO
IFC '"l','FSINCOS' IS INSTRUCTION FSINCOS
IFC '"S" IS INDEXING PART OF THE ADDR.MODE
MOVE.W #$58000+(\3<\7)+(\3+\1,MC68881+COMMAND MEM. TO REG. OPERATION
\NULCA CMPI #$89000,MC68881+RESPONSE READ RESPONSE REGISTER
BEQ.S \NULCA REREAD UNTIL EVALUATE EA AND TRANSFER
DATA
MOVE.B \2,MC68881+OPER BYTE DATA TRANSFER
\NULREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE?
BMI.S \NULREL BRANCH UNTIL NULL RELEASE
ENDC
IFNC '"l','FSINCOS' IS INSTRUCTION NOT FSINCOS
IFC '"S" IS <EA> NOT INDIRECT WITH INDEXING
MOVE.W #$58000+(\3<\7)+(\3+\1,MC68881+COMMAND MEM. TO REG. OPERATION
\NULCA CMPI #$89000,MC68881+RESPONSE READ RESPONSE REGISTER
BEQ.S \NULCA REREAD UNTIL EVALUATE EA AND TRANSFER
DATA
MOVE.B \2,MC68881+OPER MOVE DATA INTO OPERAND REGISTER
\NULREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE?
BMI.S \NULREL BRANCH UNTIL NULL RELEASE
ENDC
IFNC '"l','FSINCOS' IS INSTRUCTION NOT FSINCOS
IFC '"S" IS <EA> NOT INDIRECT WITH INDEXING
MOVE.W #$58000+(\3<\7)+(\3+\1,MC68881+COMMAND MEM. TO REG. OPERATION
\NULCA CMPI #$89000,MC68881+RESPONSE READ RESPONSE REGISTER
BEQ.S \NULCA REREAD UNTIL EVALUATE EA AND TRANSFER
DATA
MOVE.B \2,\3,MC68881+OPER BYTE DATA TRANSFER
\NULREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE?
BMI.S \NULREL BRANCH UNTIL NULL RELEASE
ENDC
ENDC
WID ADDRESSING MODES: DN, (AN)+, -(AN), D(AN), D(AN,IX) 
XXX.W, XXX.L, (D,PC), D(PC,IX)
******************************************************************************
~GB MACRO
IFC '"l'," IS <EA> INDIRECT WITH INDEXING
MOVE.W #$5800+(\3<\7)+(\3+\1,MC68881+COMMAND MEM. TO REG. OPERATION
\NULCA CMPI #$89000,MC68881+RESPONSE READ RESPONSE REGISTER
BEQ.S \NULCA REREAD UNTIL EVALUATE EA AND TRANSFER
DATA

INSTRUCTION = FP INSTRUCTION NUEMONIC (I.E. FADD)

WHERE: INSTRUCTION= FP INSTRUCTION NUEMONIC (I.E. FADD)

<EA>= SOURCE ADDRESSING MODE

FPN= DESTINATION REGISTER

NO REGISTERS MODIFIED OR DESTROYED!

VALID ADDRESSING MODES: Dn, (AN)+, -(AN), D(AN), D(AN,IX)

MEMREG MACRO

WHERE: INSTRUCTION= FP INSTRUCTION NUEMONIC (I.E. FADD)

<EA>= SOURCE ADDRESSING MODE

FPN= DESTINATION REGISTER

NO REGISTERS MODIFIED OR DESTROYED!

VALID ADDRESSING MODES: Dn, (AN)+, -(AN), D(AN), D(AN,IX)

*****************************************************************************

MOTOROLA AN947/D
523 MEMREGD MACRO
524 IFC \"4\", IS <EA> INDIRECT WITH INDEXING
525 MOV.E.W #$48000+(3<<7)+1,MC68881+OPER MOV.E.W #$4400+(3<<7)+1,MC68881+COMMAND MEM. TO REG. OPERATION
526 \ANULCA CMPI #$89000,MC68881+RESPONSE READ RESPONSE REGISTER
527 BEQ.S \ANULCA REREAD UNTIL EVALUATE EA AND TRANSFER DATA
528 * MOV.E.L \(2,MC68881+OPER SINGLE PRECISION DATA TO FP REG.
529 \ANULREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE?
530 BMI.S \ANULREL BRANCH UNTIL NULL RELEASE
531 ENDC
532
533 IFNC \"4\", IS <EA> NOT INDIRECT WITH INDEXING
534 MOVE.W #$6400+(4<<7)+1,MC68881+COMMAND MEM. TO REG. OPERATION
535 \ANULCA CMPI #$89000,MC68881+RESPONSE READ RESPONSE REGISTER
536 BEQ.S \ANULCA REREAD UNTIL EVALUATE EA AND TRANSFER DATA.
537 * MOV.E.L \(2,3,MC68881+OPER SINGLE PRECISION DATA TO FP REG.
538 \ANULREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE?
539 BMI.S \ANULREL BRANCH UNTIL NULL RELEASE
540 ENDC
541 ENDM
542
543 *******************************************************
544 * MC68881 DOUBLE PRECISION VALUE MEMORY TO FP-REG. OPERATION
545 * MEMREGD INSTRUCTION, <EA>, FPN
546 * WHERE: INSTRUCTION= FP INSTRUCTION NEMONIC (I.E. FADD)
547 * <EA>= SOURCE ADDRESS REGISTER, SURROUNDED BY PARENTHESES,
548 * CONTAINING THE PREVIOUSLY ENTERED ADDRESSING MODE
549 * (I.E. (AN)).
550 * FPN= DESTINATION REGISTER
551 * NO REGISTERS MODIFIED OR DESTROYED!
552 * VALID ADDRESSING MODES: (AN)
553 *
554 *******************************************************
555 MEMREGD MACRO
556 MOVE.W #$6400+(3<<7)+1,MC68881+COMMAND MEM. TO REG. OPERATION
557 \ANULCA CMPI #$89000,MC68881+RESPONSE READ RESPONSE REGISTER
558 BEQ.S \ANULCA REREAD UNTIL EVALUATE EA AND TRANSFER DATA.
559 * MOV.E.L \(2,MC68881+OPER HIGH ORDER LONG WORD
560 MOV.E.L \(2,MC68881+OPER LOW ORDER LONG WORD
561 \ANULCA TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE?
562 BMI.S \ANULREL BRANCH UNTIL NULL RELEASE
563 ENDC
564
565 *******************************************************
566 * MC68881 EXTENDED PRECISION VALUE MEMORY TO FP-REG. OPERATION
567 * MEMREGX INSTRUCTION, <EA>, FPN
568 * WHERE: INSTRUCTION= FP INSTRUCTION NEMONIC (I.E. FADD)
569 * <EA>= SOURCE ADDRESS REGISTER, SURROUNDED BY PARENTHESES,
570 * CONTAINING THE PREVIOUSLY ENTERED ADDRESSING MODE
571 * (I.E. (AN)).
572 * FPN= DESTINATION REGISTER
573 * NO REGISTERS MODIFIED OR DESTROYED!
574 * VALID ADDRESSING MODES: (AN)
575 *
576 *******************************************************
577 MEMREGX MACRO
578 MOVE.W #$6400+(3<<7)+1,MC68881+COMMAND MEM. TO REG. OPERATION
589 \&NULCA CMP  #88H,MC68881+RESPONSE READ RESPONSE REGISTER
590 BEQ.S \&NULCA REREAD UNTIL EVALUATE EA AND TRANSFER
591 *
592 MOVE.L \2,MC68881+OPER HIGH ORDER LONG WORD
593 MOVE.L 4\2,MC68881+OPER MID-ORDER LONG WORD
594 MOVE.L 8\2,MC68881+OPER LOW ORDER LONG WORD
595 \&NULREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE?
596 BMI.S \&NULREL BRANCH UNTIL NULL RELEASE
597 ENDM
598
599******************************************************************************
600 * MC68881 PACKED BCD VALUE MEMORY TO FP-REG. OPERATION
601 *
602 * MEMREGP INSTRUCTION, <EA>, FPN
603 *
604 * WHERE: INSTRUCTION = FP INSTRUCTION NUEMONIC (I.E. FADD)
605 * <EA> = SOURCE ADDRESS REGISTER, SURROUNDED BY PARENTHESIS,
606 * CONTAINING THE PREVIOUSLY ENTERED ADDRESSING MODE
607 * (I.E. (AN)).
608 * FPN = DESTINATION REGISTER
609 *
610 * NO REGISTERS MODIFIED OR DESTROYED!
611 *
612 * VALID ADDRESSING MODES: (AN)
613 *
614 *
615******************************************************************************
616 MEMREGP MACRO
617 MOVE.W #$4000\(<\3<<7>+\1,MC68881+COMMAND MEM. TO REG. OPERATION
618 \&NULCA CMP  #88H,MC68881+RESPONSE READ RESPONSE REGISTER
619 BEQ.S \&NULCA REREAD UNTIL EVALUATE EA AND TRANSFER
620 *
621 MOVE.L \2,MC68881+OPER HIGH ORDER LONG WORD
622 MOVE.L 4\2,MC68881+OPER MID-ORDER LONG WORD
623 MOVE.L 8\2,MC68881+OPER LOW ORDER LONG WORD
624 \&NULREL TST.B MC68881+RESPONSE IS RESPONSE NULL RELEASE?
625 BMI.S \&NULREL BRANCH UNTIL NULL RELEASE
626 ENDM
627
628******************************************************************************
629 * MC68881 FP-REG. TO FP-REG. OPERATION
630 *
631 * MEMREG INSTRUCTION, FPN, FPN, FNQ
632 *
633 * WHERE: INSTRUCTION = NUEMONIC FOR THE FP INSTRUCTION (I.E. FADD)
634 * FPN = FP SOURCE REGISTER
635 * FPN = FP DESTINATION REGISTER
636 * FNQ = SECOND FP DESTINATION REGISTER FOR FSINCOS
637 *
638 * NO REGISTERS MODIFIED OR DESTROYED!
639 *
640******************************************************************************
641 ICMP MACRO
642 IF INSTR. IS FSINCOS DO THIS ROUTINE
643 MOVE.W \(<\2<<10>+\(<\4<<7>+\3\1,MC68881+COMMAND REG. TO REG. FSINCOS
644 \&NULCA TST.B MC68881+RESPONSE READ RESPONSE REGISTER
645 BMI.S \&NULCA REREAD UNTIL NULL RELEASE (CA=0)
646 ENDC
647 IFNC \(<\1\', 'FSINCOS' ROUTINE FOR ALL OTHER ARITHMETIC INSTRS.
648 MOVE.W \(<\2<<10>+\(<\3<<7>+\1,MC68881+COMMAND REG. TO REG. OPERATION
649 \&NULCA TST.B MC68881+RESPONSE READ RESPONSE REGISTER
650 BMI.S \&NULCA REREAD UNTIL NULL RELEASE (CA=0)
651 ENDC
652 ENDM
**MC68881 CONSTANT IN ROM TO FP-REG. OPERATION**

**FMOVEOM MACRO CC, FPN**

WHERE:
CC = MC68881 CONSTANT

$00 PI
$0B LOG10(2)
$0C E
$0D LOG2 (E)
$0E LOG10(E)
$0F 0.0
$30 LOG(2)
$31 LOG(10)
$32 10^0
$33 10^1
$34 10^2
$35 10^4
$36 10^8
$37 10^16
$38 10^32
$39 10^64
$3A 10^128
$3B 10^256
$3C 10^512
$3D 10^1024
$3E 10^2048
$3F 10^4096

FPN = FP DESTINATION REGISTER

NO REGISTERS MODIFIED OR DESTROYED!

**FMOVEOM MACRO**

MOVE.W #65536+4($2<7)+V/1, MC68881+COMMAND REG. TO REG. OPERATION

\$NULCA TST.B MC68881+RESPONSE READ RESPONSE REGISTER

EML.S &NULCA REREAD UNTIL NULL RELEASE (CN=0)

**ENDM**

**MC68881 CONDITIONAL BRANCH**

**FBCC <SIZE> CONDITION, ADDRESS**

WHERE:
<SIZE> ALLOWABLE BRANCH SIZES

CONDITION= CC, THE FLOATING POINT CONDITION (I.E. GT)

ADDRESS= BRANCH ADDRESS

REGISTERS MODIFIED OR DESTROYED: 0 1 2 3 4 5 6 7

D X

A

**FBCC MACRO**

MOVE.W #1, MC68881+COND BEGIN COPROCESSOR COMMUNICATION

\$NOPASS MOVE.W MC68881+RESPONSE, D0 IS CN-BIT SET

EML.S \$NOPASS REREAD UNTIL NULL RELEASE (CN=0)

BTST #TBIT, D0 IS CONDITION TRUE

BNE. \$0 \#2 BRANCH IF CONDITION TRUE!

**ENDM**
**MC68881 TEST FP CONDITION, DECREMENT, AND BRANCH**

**FDBCC**  
CONDITION, DN, ADDRESS

WHERE: CONDITION= CC, FLOATING POINT CONDITION  
DN= MAIN PROCESSOR DATA REGISTER TO BE DECREMENTED  
ADDRESS= BRANCH ADDRESS

REGISTERS MODIFIED OR DESTROYED: 0 1 2 3 4 5 6 7  
D X  
A

**FDBCC MACRO**

MOVE.W "+1,MC68881+COND  
\$NOPASS MOVE.W MC68881 RESPONSE,DO  
\$NOPASS \nBML.S \$NOPASS REREAD UNTIL NULL RELEASE (CA=0)  
\BIST #TBIT,DO  
\DBNE \’2,\’3  
EQUALS -1

ENDM

**FSCC**  
CONDITION, ADDRESS

WHERE: CONDITION= CC, FLOATING POINT CONDITION  
ADDRESS= BRANCH ADDRESS

REGISTERS MODIFIED OR DESTROYED: 0 1 2 3 4 5 6 7  
D X  
A

**FSCC MACRO**

MOVE.W "+1,MC68881+COND  
\$NOPASS MOVE.W MC68881 RESPONSE,DO  
\$NOPASS \nBML.S \$NOPASS REREAD UNTIL NULL RELEASE (CA=0)  
\BIST #TBIT,DO  
\SNE \’2  
SET BYTE AT POINTER\’2 TO 1\’S IF  
\CONDITION TRUE, IF CONDITION FALSE  
\SET BYTE TO 0\’S

ENDM

**MC68881 FP MOVE MULTIPLE COPROCESSOR REGISTERS TO MEMORY**

FMVEMSM FPR0,FPR1,FPR2,FPR3,FPR4,FPR5,FPR6,FPR7,<EA>,PREDECREMENT

WHERE: FPR0= (FP REG. #0) 1 IF SELECTED, 0 IF NOT  
FPR1= ( " #1) " "  
FPR2= ( " #2) " "  
FPR3= ( " #3) " "  
FPR4= ( " #4) " "  
FPR5= ( " #5) " "  
FPR6= ( " #6) " "  
FPR7= ( " #7) " "  
<EA>= DESTINATION ADDRESSING MODE  
PREDECREMENT= Y (IF PREDECREMENT MODE IS BEING USED), OR  
N (IF OTHER MODE IS BEING USED).  
REGISTERS MODIFIED OR DESTROYED: 0 1 2 3 4 5 6 7  
A X
* D X X X X *
* VALID ADDRESSING MODES: AN, -(AN), D(AN), D(AN, IX) *
* XXX.W, XXX.L *
* *
* ***************************************************************************
* PMOVEFM MACRO *
* IFC "V", "Y" IS THE ADDRESSING MODE PREDECREMENT *
* *
* THIS CODE IS FOR PREDECREMENT ADDRESSING MODE *
* *
* MOVE.W #0E000+4\8\7\6\5\4\3\2\1, MC68681+COMMAND CP REGISTER BIT *
* 8NULCA CMPI #8900, MC68681+RESPONSE READ RESPONSE REGISTER *
* BEQ.S 8NULCA REREAD UNTIL TRANSFER MULTIPLE REGS. *
* *
* THIS CODE CALCULATES THE TOTAL # OF REGISTERS TO BE TRANSFERRED *
* *
* MOVEQ \|1\+\|2\+\|3\+\|4\+\|5\+\|6\+\|7\+\|8\-1, D3 *
* *
* TST.W MC68681+REGSEL READ REGISTER RESPONSE REGISTER *
* MOVE.L MC68681+OPER, AO LOAD ADDRESS OF THE OPERAND REG. *
* \8AGAIN MOVE.L (AO), D0 LOAD HIGH ORDER WORDS *
* MOVE.L (AO), D1 LOAD MID ORDER WORDS *
* MOVE.L (AO), D2 LOAD LOW ORDER WORDS *
* MOVEM.L D0-D2, \9 * STACK HIGH ORDER WORD IN LOW ORDER *
* DDBA D3, \8AGAIN MEMORY AND LOW ORDER WORD IN HIGH *
* HAVE ALL REGISTERS BEEN TRANSFERRED ORDER MEMORY *
* *
* \8NULSEL TST.B MC68681+RESPONSE IS RESPONSE NULL RELEASE? *
* BMI.S \8NULSEL BRANCH UNTIL NULL RELEASE *
* ENDC *
* IFC "V", "N" IS ADDRESSING MODE NOT PREDECREMENT *
* ***************************************************************************
* *
* THIS CODE IS FOR ALL VALID ADDRESSING MODES OTHER *
* THAN PREDECREMENT *
* *
* ***************************************************************************
* MOVE.W \$E000+4\12\3\1\5\6\7\8, MC68681+COMMAND CP REGISTER BIT *
* 8NULCA CMPI #8900, MC68681+RESPONSE READ RESPONSE REGISTER *
* BEQ.S 8NULCA REREAD UNTIL TRANSFER MULTIPLE REGS. *
* MOVEM.L \#(\|8\+\|7\+\|6\+\|5\+\|4\+\|3\+\|2\+\|1\)\*3-1, D0 COUNT REG. FOR DDBA STM. *
* TST.W MC68681+REGSEL READ REGISTER RESPONSE REGISTER *
* \8AGAIN MOVE.L MC68681+OPER, (AO) LOAD DATA ON TO THE STACK *
* DDBA D0, \8AGAIN LOOP UNTIL ALL DATA IS LOADED *
* \8NULSEL TST.B MC68681+RESPONSE IS RESPONSE NULL RELEASE? *
* BMI.S \8NULSEL BRANCH UNTIL NULL RELEASE *
* ENDC *
* ENDM
.Move.W #0000+\2\3\4\5\6\7\8\9,MC68881+COMMAND register bit.

\NULCA CNTL #8900,MC68881+RESPONSE REG.

Movi.W #\2,MC68881+RESPONSE.

\NULCA CNTL #0000,MC68881+RESPONSE.

Movi.W #0000+\2\3\4\5\6\7\8\9,MC68881+COMMAND register bit.

\NULCA CNTL #0000,MC68881+RESPONSE.

Movi.W #\2,MC68881+RESPONSE.

\NULCA CNTL #0000,MC68881+RESPONSE.

Movi.W #0000+\2\3\4\5\6\7\8\9,MC68881+COMMAND register bit.

\NULCA CNTL #0000,MC68881+RESPONSE.
MC68881 FP MOVE FROM CONTROL/STATUS/INSTRUCTION ADDRESS REGISTER

MOVE.W \d1\+2000,MC68881+COMMAND
CMP.W \d56000,MC68881+RESPONSE
BEQ.S \dNULCA
MOVE.L \d1\d12,MC68881+OPER
\dNULREL TST.B MC68881+RESPONSE
BMI.S \dNULREL
ENDC
IFNC \d3,
MOVE.W \d1\d12000,MC68881+COMMAND
CMP.W \d56000,MC68881+RESPONSE
BEQ.S \dNULCA
MOVE.L \d1\d12,MC68881+OPER
\dNULREL TST.B MC68881+RESPONSE
BMI.S \dNULREL
ENDC

MC68881 FSAVE THE INTERNAL OF THE MACHINE

THIS IS A PRIVILEGED INSTRUCTION WHICH IS GENERALLY ONLY USED
IN THE OPERATING SYSTEM FOR CONTEXT SWITCHING!

FSAVEST <EA>
WHERE: <EA> = PREDECREMENT MODE - (AN)

REGISTERS MODIFIED OR DESTROYED: 0 1 2 3 4 5 6 7
A X
D X X

VALID ADDRESSING MODES: - (AN)
FSAVEST MACRO
@START MOVE.W MC68881SAVE, DO READ THE SAVE REGISTER
MOVE.W DO, D1 MAKE A COPY OF THE FORMAT WORD
ANDI.W #$FF00, D1 ISOLATE THE FORMAT WORD
CMPI.W #$0100, D1 IF NULL IDLE, NO STATE SAVE
BEQ.S @START KEEP CHECKING UNTIL CP IS FINISHED
processing
LEA MC68881+OPER, A0 LOAD OPERAND REGISTER TO A0
MOVE.B D0, D1 THE LENGTH OF THE DATA TO BE TRANSFERRED
LSR.B #2, D1 DEVIDE BY 2 TO ADJUST FOR WORD TRANSFER
EXT.W D1 ESTABLISH COUNTER AS A WORD FOR DBRA
SUBQ.W #1, D1 D1= COUNTER FOR DBRA
@LOAD MOVE.L (AO), D1 STORE THE INVISIBLE STATE
@LOAD DBRA D1, @LOAD REPEAT UNTIL ALL DATA IS TRANSFERRED
@NULL SWAP D0 PLACE FORMAT WORD IN UPPER 16 BITS OF DO
MOVE.L D0, D1 STORE FORMAT WORD ON THE STACK
ENDM

******************************************************************************

FSAVEST MACRO
@START MOVE.W MC68881+SAVE, DO READ THE SAVE REGISTER
MOVE.W DO, D1 MAKE A COPY OF THE FORMAT WORD
ANDI.W #$FF00, D1 ISOLATE THE FORMAT WORD
CMPI.W #$0100, D1 IF NULL IDLE, NO STATE SAVE
BEQ.S @START KEEP CHECKING UNTIL CP IS FINISHED
processing
LEA MC68881+OPER, A0 LOAD OPERAND REGISTER TO A0
MOVE.B D0, D1 THE LENGTH OF THE DATA TO BE TRANSFERRED
LSR.B #2, D1 DEVIDE BY 2 TO ADJUST FOR WORD TRANSFER
EXT.W D1 ESTABLISH COUNTER AS A WORD FOR DBRA
SUBQ.W #1, D1 D1= COUNTER FOR DBRA
@LOAD MOVE.L (AO), D1 STORE THE INVISIBLE STATE
@LOAD DBRA D1, @LOAD REPEAT UNTIL ALL DATA IS TRANSFERRED
@NULL SWAP D0 PLACE FORMAT WORD IN UPPER 16 BITS OF DO
MOVE.L D0, D1 STORE FORMAT WORD ON THE STACK
ENDM

******************************************************************************

FSAVEST MACRO
@START MOVE.W MC68881+SAVE, DO READ THE SAVE REGISTER
MOVE.W DO, D1 MAKE A COPY OF THE FORMAT WORD
ANDI.W #$FF00, D1 ISOLATE THE FORMAT WORD
CMPI.W #$0100, D1 IF NULL IDLE, NO STATE SAVE
BEQ.S @START KEEP CHECKING UNTIL CP IS FINISHED
processing
LEA MC68881+OPER, A0 LOAD OPERAND REGISTER TO A0
MOVE.B D0, D1 THE LENGTH OF THE DATA TO BE TRANSFERRED
LSR.B #2, D1 DEVIDE BY 2 TO ADJUST FOR WORD TRANSFER
EXT.W D1 ESTABLISH COUNTER AS A WORD FOR DBRA
SUBQ.W #1, D1 D1= COUNTER FOR DBRA
@LOAD MOVE.L (AO), D1 STORE THE INVISIBLE STATE
@LOAD DBRA D1, @LOAD REPEAT UNTIL ALL DATA IS TRANSFERRED
@NULL SWAP D0 PLACE FORMAT WORD IN UPPER 16 BITS OF DO
MOVE.L D0, D1 STORE FORMAT WORD ON THE STACK
ENDM

******************************************************************************
AN947/D

1033 MOVE.W #$0000, MC68881+COND
1034 START MC68881+RESPONSE
1035 BMI & $NOPAS
1036 ENDM

FNP COMMAND TO FP REG.
TEST RESPONSE

ARCHIVE DOCUMENT - NOT FOR NEW DESIGN